2002

Random search of AND-OR graphs representing finite-state models

David Robert Owen
West Virginia University

Random Search of AND-OR Graphs Representing Finite-State Models

David R. Owen

Thesis Submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Bojan Cukic, Ph.D., Chair
Tim Menzies, Ph.D.
Mark Shereshevsky, Ph.D.
Krishnamurthy Subramani, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2002

Keywords: Software Verification, Model Checking, Testability,
Random Search, AND-OR Graph, Finite-State Machine

Abstract

Random Search of AND-OR Graphs Representing Finite-State Models

David R. Owen

Model checking tools have been effective in testing concurrent software represented by communicating finite-state machines. But these tools may require a very large amount of memory. A finite-state model can be translated automatically into a compact AND-OR graph. We use an abductive random search scheme to extract, from the AND-OR graph, information about the execution of the program represented by the original finite-state model.

We use the search to measure *testability*. For AND-OR graphs representing highly testable programs, we find quickly everything it is possible to find; that is, if the number of unique goals found is plotted, we see a quick rise to a level plateau. The search can also be used to prove simple logical properties.

To determine what makes a finite-state model more or less testable, we analyze random search results for 15,000 randomly generated models with a range of attributes. We also show how this technique can be used on a model much too large for model checking tools.

Contents

List of Figures

Chapter 1

Introduction

We would like to produce correct software as quickly and as cheaply as possible. Often the majority of time and resources required by a project is spent finding and correcting errors. This is true for the individual programming student; this is true for a large team of experienced professionals. As projects grow large and complex, as correctness grows more critical, and as more people are involved in the project, it becomes extremely important and extremely difficult to find and correct errors [18].

Designers use models—prose, flow charts, pseudo-code, formal mathematical language—to document and communicate about projects. At the low level of the actual code, even for the people who wrote the code, it is often extremely difficult to reason clearly about the behavior of a program. For program models written in formal mathematical language, there are powerful and established testing tools; they are (appropriately) called "model checkers" (see page 10 for more information and references concerning the development and use of model checkers).

The type of model checked by these tools is called a *finite-state machine* (or more specifically, a *set of communicating finite-state machines*; see page 12). In this paper we present an alternative to model checking—a new technique for testing finite-state machine software models. We believe that our alternative technique might eventually be used to prove properties in the same way model checking tools do now, but the work presented in this thesis is primarily concerned with extracting information about the *testability* of systems represented by finite-state machines.

Figure 1.1 shows the basic intuition behind testability claims and experiments presented in this thesis. The plot marked *good* quickly rises as many goals are reached early and then levels of, indicating that the search can no longer find anything new. The plot marked *best* rises more quickly to a higher plateau. We infer from this that the *best* model is more testable

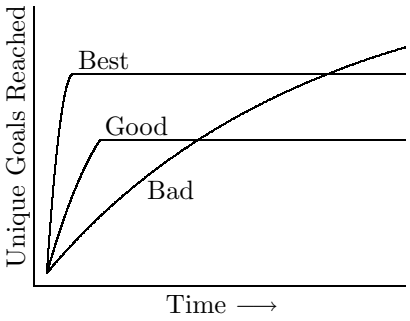


Figure 1.1: Inferring Testability from Partial Search.

than the *good* model: fewer tests are required, and more goals are reached. The plot marked *bad* never levels off, which means that the model will be difficult to test: even after many tests, we keep finding more information—there is no way to tell when we would be able to stop testing.

Model checking tools perform a complete search of all program behavior represented by the model. Our technique performs only a partial search, which means that only a portion of the total program behavior represented by the model is explored. We believe that our technique will prove to be a legitimate alternative to (often intractable) exhaustive search techniques, and argue for the value of the partial search in various ways (see page 6). Our central assumption is that most real programs are structured so that just a few key variables largely determine the behavior of the entire program. For these programs, we expect to see plateau shapes like those marked *good* and *best* above, because the random search will quickly find the key variables and therefore everything else it is capable of finding. We justify this assumption about program structure by a series of experiments yielding the anticipated plateau shape.

One way in which model checking and our search technique are alike is that they are both automatic. This is important because a human being checking a program (or a model of a program) is biased; people check where they expect to find errors. Unbiased automatic tools help us detect hard-to-find, unexpected errors. Model checkers are unbiased in the sense that they look everywhere, not just where errors are expected. Our partial search is unbiased because it is random; that is, the portion of the model checked depends on a series of random choices that do not reflect any intuition about where to look for errors.

In addition to our work on testability, we also show how our search method might be

used to verify temporal properties of models (testability would still come into play in this work as a measure of confidence in the specific verification result). This is an exciting area for future work, because our random search of AND-OR graphs does not require the large amount of memory used by model checkers, which means that we may be able to verify much larger models in the future.

Chapter 2

Literature Review

Here we review work in the following areas: the importance of software measurement in general and testability in particular (2.1), the success of partial abductive random search in knowledge-based systems (2.2), program structure amenable to random search (2.3), program verification by model checking (2.4), and the *communicating finite-state machine* form used in model checking and in our work to represent programs whose processes execute in parallel and interact via shared resources (2.5).

2.1 Software Measurement and Testability

In any engineering discipline, accurate and meaningful measurement is the key to controlling projects, choosing among alternative strategies, and improving quality over time [5]. This is obviously true in engineering disciplines involving the physical sciences, but has been controversial in software engineering. Many believe that "important software attributes like dependability, quality, usability and maintainability are simply not quantifiable" [8].[1] On the other hand, there are those who believe that, if the most important attributes are unmeasurable, we should be unsatisfied with the state of the art. We ought to use what ever imperfect measurement techniques are available; we should "try to use measurement to advance our understanding" [8] of software attributes and develop better measuring techniques in the process [5, 8].

The cost (in time and money) of software projects is notoriously hard to predict and control; all projects are likely to go over budget and beyond deadlines—the question is, how far over budget? How far beyond deadlines? [1, 5, 8] Also, re-organizing existing projects and

[1]Note that in [8] the same point is being made that we make here; we have simply re-worded the beginning of the sentence ("Many believe that ...").

adding new staff usually causes projects to cost even more time and money [9], which gives us all the more reason for planning properly from the beginning.

Friedman and Voas make the point adamantly that "software assessment" is not "development process assessment" [18]. While the latter may be important, and while it may be the best predictor of future software quality, here we focus (as they do) on assessing software products. From initial need to delivered product to upgrade or replacement, software grows through the stages of a *life cycle*. In general the concern of software developers shifts from achieving quality to assessing quality. Surprisingly, it often costs even more to assess quality than to achieve it in the first place [18]. Because of this, the *testability* (a measure of how difficult it is to assess the quality) of software is extremely important.

In practice, testability is often viewed simply as being inversely proportional to the complexity of a program [18]; the more complex the program, the more difficult it will be to test. A more useful but still intuitive definition is "the probability that a program will fail under test if it contains at least one fault" [3]. Our view is consistent with this definition, if the following assumption is granted: the greater the proportion of a program's execution space exercised by testing, the more likely a fault will be found and cause the test to fail (if some fault exists). That is, a highly testable program is one for which a relatively small number of tests tell us a relatively large amount. This has been called the "reachability view" of program testing and testability [27].

Consider a graph representing all possible executions of a program, with a fault as some location in the graph, and a test execution as some pathway through the graph. Testability can be thought of as the probability that the pathway reaches the fault. In this thesis we present a technique for measuring how much of a program graph can be reached by a series of test pathways and how quickly that happens (the sum of the length of the pathways). For our purposes, *testability* involves these two factors: (1) the amount of the program reached by our tests and (2) the number of tests required to reach it. Using these values as indicators of testability, we will also attempt to show how the structure of programs may in some cases be manipulated to make them more testable.

Software measurements should avoid subjectivity; that is, the value of the measurement should be, as much as possible, independent of the person or machine doing the measurement [5]. For example, a popular and apparently straight-forward measurement of a program is the number of lines of code. Clearly there has to be agreement about which lines should be counted (comments? blank lines? lines copied from another source?) in order to make *lines of code* a useful, objective measure.

In the work presented here testability measurements are susceptible to subjectivity only in the creation of the finite-state machine model representing the program. This is something that might eventually be automated, as much work has been done in the model checking community to produce finite-state models automatically from source code [14, 21].

Karoui et.al. have proposed a set of formally defined parameters influencing the testability of finite-state machine models [16, 17]: "controllability," "fuzziness," "state-characterization degree," "abstraction degree"—they provide formulas for determining each of these attributes. The following (exaggerated) analogy captures the difference between their approach and ours: their approach requires a great deal of information about the finite-state machine implementation. It is like measuring the kinetic energy of a billion molecules and adding it all up in order to determine the temperature of a liquid. Our approach is to just use a simple thermometer capable of comparing the temperature of one liquid to another.

2.2 The Success of Random Search

The experimental results from which we infer testability measurements are based on an automatic partial random search of the program model. The random search presented here is based on a technique called *abduction* or *hypothesis testing* that has been used to evaluate knowledge-based systems. Abduction runs on an AND-OR graph representing what is usually a *vague domain*—the model contains contradictions and gaps that reflect experts' disagreements; or, even if the information is available and the experts agree, it may be too expensive to build the best possible model [32, 33]. Abduction is a way of testing these imperfect theory models; that is, extracting from them competing hypotheses that can be ranked by quality. The best hypothesis is the hypothesis with the most predictive power, and the best model is the model with the best hypotheses.

Menzies and Cukic argue that software testing is fundamentally a search process [25, 26]. For example, abduction involves searching for pathways through an AND-OR graph and checking for contradictions, model checking tools search through a finite space for states representing counter examples to desired properties, etc. In general, the goal of software testing is to find a consistent tree of pathways through the program execution space. If some leaf represents an undesirable *fault,* the tree shows where the problem came from and hopefully leads us to a modification that could prevent it [28, 29].

Menzies et.al. have implemented a generalized abductive hypothesis testing procedure called HT4, which performs an exhaustive search (requiring exponential time) for internally

consistent *worlds* implicit in an AND-OR graph [25,30,32]. While the total AND-OR graph may contain inconsistent information leading to competing hypotheses, within each individual world everything is consistent. In order to find the best world (the world whose hypotheses have the most predictive power), HT4 finds every possible world in the AND-OR graph. To evaluate HT4, Menzies et.al. created another procedure, "HT4-dumb," which simply chooses a single world (instead of looking for the best one). Surprisingly, HT4-dumb performed nearly as well as HT4. This prompted the development of a randomized hypothesis testing procedure, HT0. HT0 is able to do in approximately $O(n^2)$ time what takes HT4 exponential time; in addition, HT0 works on models much too large for HT4 [25,30].

For our purposes, *random* testing means search from some (not necessarily random) input through a space with contradictions; when two contradictory options are encountered, the choice between them is made at random. Others primarily concerned with the statistical significance of test results sometimes characterize random testing differently: a deterministic program is fed inputs chosen at random from a distribution called an *operational profile* [23][2]. This approach should not be confused with ours.

To explain the success of the randomized testing procedure, Menzies et.al. claim that, in general, "probes into software rapidly saturate" [30]; that is, individual tests at first yield a great deal of information, but as testing progresses its efficiency quickly tapers off; more and more tests yield less and less new information. This is why HT4, even though it works so much harder and longer, cannot find any more than HT0. Beyond the point of saturation, additional testing is of little practical value. But how quickly is the point of saturation reached? Menzies et.al. cite a variety of sources and conclude that "the average size and complexity of the used portions of our programs is much smaller than we might think" [30].

2.3 The Right Program Structure for Random Search

Software testing, then, is (arguably) equivalent to a search for the interesting portions of a space representing the execution of the program. And if the structure of the program's interesting portions is simple, as Menzies et.al. argue it is in most cases, a few random probes will lead quickly to the interesting portions of the program model [26]. In addition to the HT4 vs. HT0 comparison, Menzies and Cukic cite Crawford and Baker's comparison of an exhaustive depth-first search backtracking algorithm, TABLEAU, to ISAMP, a random

[2]Hamlet also mentions that to some people *random* testing is "haphazard," "not well organized," or even "gratuitously wrong" testing; that's *not* what we're talking about here [23].

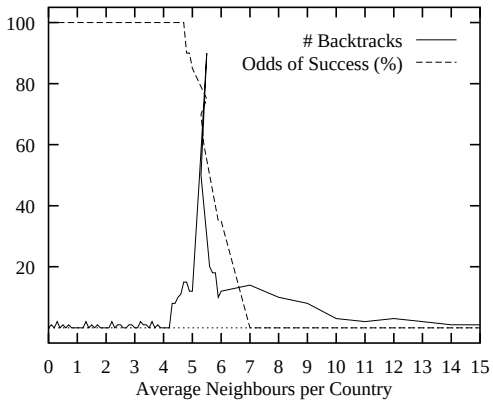


Figure 2.1: For a graph-coloring problem, a *spike* in the time required appears at the transition between instances solved quikcly and instances that cannot be solved.

search-based theorem prover. The two tools were run on a set of scheduling problems based on real-world parameters. ISAMP not only worked much more quickly but also found more solutions (possible schedules) than TABLEAU. Crawford and Baker offer the following explanation for the success of ISAMP: the test problems contain a large proportion of "dependent" variables whose values are determined by a small number of "control" variables [15, 25, 26].

Kautz and Selman compared TABLEAU to Graphplan, a systematic (and exhaustive) search tool designed specifically for planning problems, and then compared TABLEAU and Graphplan to Walksat, a random search tool [12]. On difficult planning problems Walksat far outperformed the systematic alternatives. Eventually Kautz and Selman decided to use Walksat to find a solution and TABLEAU to check whether the solution was optimal (TABLEAU was unable to find the optimal solution directly in a reasonable amount of time).

If many programs fall into the *quick-saturation* (high testability) category, what about those that don't? How do we know whether a particular program has the right structure for quick random search? How do we design a new program so that it will have the right structure? Menzies and Cukic cite Cheeseman et.al., arguing that for NP-hard problems like the exhaustive search performed by HT4 or TABLEAU (problems requiring exponential time) the actual problem instance will with high probability fall into one of two categories: it will either be very easy or very hard—either we can solve it quickly or see quickly that we cannot solve it [22, 26]. Figure 2.1 shows a graph illustrating this *phase transition.* Only in a narrow middle range of problem instances does the plotted time required show a high *spike.* For the majority of NP-hard problems, including our general testing problem formulated as

AND-OR graph search, the randomized approach will be able to quickly tell us into which of the two categories the problem falls—whether the problem is easy or too hard. So we should be able to use the random search to find out very quickly whether the program is highly testable or very difficult to test.

Menzies and Cukic argue elsewhere that "standard models of test suite sizes are gross overestimates" because they do not consider the structure of the program (which is usually simple) or the power of randomized search [25]. They cite examples of an apparently very complex natural language processing system and a large study of thousands of Fortan and C programs; in all cases the structure turned out to be surprisingly simple [28]. This is encouraging news for small-scale software projects, which may not be able to afford a long and elaborate testing regime. Even high-budget, safety-critical projects suffer from the hardware-imposed bounds on verification by model-checking tools. It is usually only the few most critical portions that are exhaustively tested. In either case it makes sense to add random search to whichever other tools are available.

The *funnel theory* of Menzies et.al. summarizes some of the conclusions and speculations of the previous four paragraphs: testing by random search terminates quickly with good results for most programs because most programs contain *funnels*—the small sets of key variables that determine the behavior of everything else [31]. This was Crawford and Baker's explanation of the success of ISAMP [15, 26]. The few key variables form a virtual funnel through which a large number of search pathways must go in order to reach an output destination. Random search quickly finds the funnel, because any number of pathways lead to it. And an exhaustive search yields little (if any) new information because all of the pathways it systematically checks lead to the same funnel. Menzies and Singh go further to show mathematically, that where funnels are present, random search will with high probability find the most *narrow* funnels—that is, the smallest sets of key variables [31]. This is just what we want when testing: the simplest explanation.

The core idea of this thesis is to apply the abductive random search technique to AND-OR graphs representing the type of finite-state models that can be verified by model-checking tools. The abductive hypothesis testing process searches for worlds of internally consistent information. We use the same technique to find a series of internally consistent assignments to a global state vector (here *internally consistent* just means that the global state vector may not assign two different local states to an individual finite-state machine from the original model). The success of the randomized version of abduction (and the interpretation of that success—that perhaps most programs have the *narrow-funnel* property) motivates

our application of these techniques to finite-state models.

2.4 Model Checking

Model checking tools use an exhaustive formal verification technique to check that a program (represented by a model) matches a specification (a set of properties) [19]. The model is written as a *finite-state concurrent system* or *system of communicating finite-state machines* (two names for the same thing, not two possible ways to write the model), and the specification is written in a form called *temporal logic* [7, 19]. Communicating finite-state machines will be described in detail in the next section (see page 12). Temporal logic is used to make statements about the time relationship between propositions without stating explicity when they occur. For example, the *globally* operator **G** might be used to say that a proposition p is true now and at all times in the future: **G** p.

Model checking has two key advantages over traditional simulation and testing techniques: (1) the procedure is fully automatic; (2) when a discrepancy is found between the model and the specification, a counterexample is generated, which makes it much easier to find and correct errors in a program [7, 11].

Unfortunately, according to Clarke et.al.:

> The main disadvantage of model checking is the *state explosion* that can occur if the system being verified has many components that can make transitions in parallel. In this case the number of global system[3] states may grow exponentially with the number of processes [7].

The general model checking technique originated in the 1980's. At that time the maximum number of states in tractable models was between 10^4 and 10^5, but since then two key developments have dramatically increased the maximum number of states: *symbolic* model checking and *partial order reduction.* In the early 1990's, researchers at Carnegie Mellon University (where Clarke and Emerson had already done pioneering work in model checking) began using binary-decision diagrams, or BDD's, to succinctly represent the global system [7]. This new *symbolic* model checking technique made it possible to check a model without ever explicity constructing its composite state graph, and models with up to 10^{20} states became tractable. Continuing work on BDD's has pushed the limit to 10^{120}.

BDD's work particularly well for representing synchronous hardware systems, but not as well for software, which is often asynchronous. In an asynchronous system several things

[3]The *global system* is a composite of individual components operating in parallel. In the next section, it will be more formally defined as a composite of communicating finite-state machines.

may be going on in parallel with no synchronizing clock, so that many different interleavings are possible. If all possible interleavings must be checked, the state space required tends to grow very large. When *partial order reduction* is used, only the interleavings relevant to the specification are checked; that is, if the specification does not care about the order of some set of events, only one interleaving of those events will be checked [7, 11].

As opposed to Clarke's symbolic model checker SMV, Holzmann's SPIN tool is designed especially for asynchronous software models [11]. SPIN employs partial order reduction and certain other optimizing techniques, and has been used in recent years to verify a range of algorithms, protocols, and system implementations. Throughout this thesis we focus on SPIN as representative of model checking tools, using examples originally written in SPIN's input language Promela (process metalanguage) and verified with SPIN (one of SPIN's nice features is an automatic translator from Promela to finite-state machine transition functions, which is the form we need for our work).

Even with the help of BDD's and partial order reduction, there are still many real systems too large to be verified by a model checker. Clarke et.al. suggest four strategies for shrinking models to a manageable size [7]:

1. **Compositional Reasoning**: the modular structure of some systems may be exploited; for example, we make certain assumptions about one part of the model in order to verify a property for another part; then we try to guarantee that the assumptions hold for the first part at any time the property must hold for the other.

2. **Abstraction**: we may be able to map the actual data values to a small set of abstract data values, for example.

3. **Symmetry**: we may be able to infer global properties from local properties in a model where many identical components interact.

4. **Induction**: if we have an *invariant* process that represents the behavior of a family of processes, we can use induction to argue that any member process has some desired property already proven for the invariant.

Here we propose a new technique for extracting information from very large finite-state models. Instead of BDD's, we use an AND-OR graph to implicitly represent the prohibitively large global state transition system, and instead of the exhaustive model checking search we use the partial random search described in the last section. We believe the new technique will eventually be able to prove temporal properties, but at this point focus on (the simpler task of) extracting testability measurements.

2.5 Communicating Finite-State Machines

Model checking tools run on software models written in various input languages all based on the mathematical notion of *communicating finite-state machines.* Here we review first the individual finite-state machine and then extend the model to a set of finite-state machines that are able to *communicate,* i.e., the behavior of a machine is influenced by the state of some other(s).

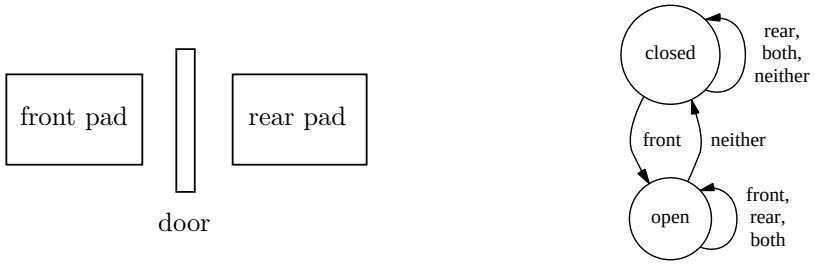


Figure 2.2: Top view of automatic door (left) and finite-state machine representing the door's function (right) from [20].

The left side of Figure 2.2 shows the top view of an automatic door (this example comes from Sipser [20]). If the door is shut and a person stands on the front pad, the door opens and remains open as long as someone is standing on either the front or rear or both pads; if the door is shut it remains shut if it senses weight on the rear pad, both pads, or neither pad. On the right is a finite-state machine diagram representing the function of the door described in the last sentence. We can think of the door controller as having two mutually exclusive states: *open* and *closed.* Transitions between these states are triggered by events in the machine's environment: whether or not there is weight on the front and rear pads. When a transition from one state to the other occurs, there is an output to the environment: when the state changes from *closed* to *open,* the door opens; when the state changes from *open* to *closed,* the door closes.

This is the typical finite-state machine model that has been around since the 1950's [6,13]. In more formal terms, it is a *Mealy* (not *Moore*) machine, a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

1. Q is a finite set of states.
2. Σ is a finite alphabet of input symbols.
3. Γ is a finite alphabet of output symbols.
4. $\delta : Q \times \Sigma \longrightarrow Q \times \Gamma$ is the transition function.

Sender

State	In	Out	Next State
q0	-	mesg0	q1
q1	ack1	-	q0
q1	ack0	-	q2
q2	-	mesg1	q3
q3	ack0	-	q2
q3	ack1	-	q0

Receiver

State	In	Out	Next State
q0	mesg1	-	q1
q0	mesg0	-	q2
q1	-	ack1	q3
q2	-	ack0	q0
q3	mesg0	-	q4
q3	mesg1	-	q5
q4	-	ack0	q0
q5	-	ack1	q3

Figure 2.3: Transition function charts for alternating bit protocol [13].

5. $q_0 \in Q$ is the initial or *start* state.

6. $F \subseteq Q$ is the set of final or *accept* states.[4]

In our simple automatic door example, Q = {open,closed}, Σ = {front, rear, both, neither}, Γ = {open the door, close the door}, etc.

Interest in finite-state machines has recently been renewed as they have been applied to problems involving communication protocols [6, 13]. The finite-state machine model is extended to a system of *communicating finite-state machines,* where individual machines share the same environment and communicate with each other via input and output sequences (one machine's output is input for the next). In order to succinctly represent real systems, the model has also been extended to include finite variables not represented directly (to represent a variable directly, we would need a machine with a separate state for every possible value of the variable). Since it is possible to convert a set of communicating finite-state machines with variables not directly represented into one larger composite machine with variables directly represented, these extensions do not alter the descriptive power of the original model, i.e., a system of communicating machines with variables is formally equivalent to the original finite-state machine definition [6, 13].

Figure 2.3 shows a set of two communicating finite-state machines, a sender machine and receiver machine, representing a version of the alternating bit protocol (this example comes from Holzmann [13]). Suppose the sender begins in state *q0.* The dash in the input column of the first transition means that the sender can move to state *q1* at any time. When the

[4]Sometimes the distinction is made between finite-state transducers, which produce output from Γ when a transition occurs, and finite-state machines, which produce no output. Also, sometimes q_0 and F are not included, or two separate input and output transition functions are included. Here we have tried to combine and summarize definitions from a few different sources ([6, 13, 20]) into a general formal definition.

sender moves to state *q1*, it outputs message *mesg0*. Message *mesg0* is then available for input to the receiver. Suppose the receiver also begins in state *q1*. The receiver will input *q0* and, based on the transition rule in line 2 of the receiver's transition function chart, move to state *q2*. From state *q2* the receiver will then move back to state *q1,* acknowledging receipt of message *mesg0* by outputting message *ack0.*

Model checking tools build the composite finite-state machine representing the interleavings of the individual communicating machines, and then exhaustively search it. In the next chapter we show how communicating finite-state machine models like this one can be automatically translated into a compact AND-OR graph. We then use a random search procedure to extract information about the equivalent finite-state composite from the AND-OR graph.

Chapter 3

Random Search of AND-OR Graphs Representing Finite-State Models

In Chapter 3, we begin by explaining how a communicating finite-state machine model can be translated into a NAYO (a type of AND-OR) graph (3.1), including here our formal definition for communicating finite-state machine systems. Next we automate the translation process and highlight important properties of NAYO graphs (3.2). We then present the most basic form of the partial random search procedure used to extract information from NAYO graphs, and show how the search might progress through a simple NAYO example (3.3).

3.1 Translating rom Finite-State Machines to NAYO (a type of AND-OR) Graphs

3.1.1 Background

Recent work by Menzies et.al. [25,27,29] shows that random search on (randomly generated) AND-OR graphs can very quickly tell us a lot about the *reachability* of those graphs. Reachability is a measure of how difficult it is to reach one place in the graph from some other. In theory, the AND-OR graphs represent real programs, and the reachability (or *unreachability*) of the AND-OR graphs represents testability of those real programs. In order to justify the claim that the AND-OR graphs represent real programs, Menzies et.al. use as an example a program model written for the model checker SPIN in the Promela language. This model was translated by hand into an AND-OR graph in order to show, for at least one very simple model, that an AND-OR graph could be created to represent the same information. The left side of Figure 3.1 shows the example written in Promela, and the right shows the translation presented by Menzies et.al. [27].

```
byte a = 1, b = 1;
bit f = 1;

active proctype A()
{
     do :: (f == 1)
           -> if :: (a == 1) -> a = 2;
                 :: (a == 2) -> a = 3;
                 :: (a == 3) -> f = 0;
                                a = 1;
              fi
     od
}

active proctype B()
{
     do :: (f == 0)
           -> if :: (b == 1) -> b = 2;
                 :: (b == 2) -> b = 3;
                 :: (b == 3) -> f = 1;
                                b = 1;
              fi
     od
}
```

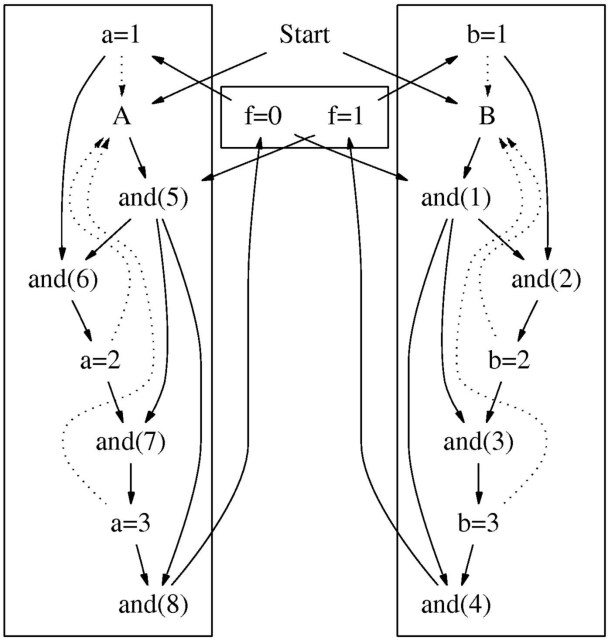
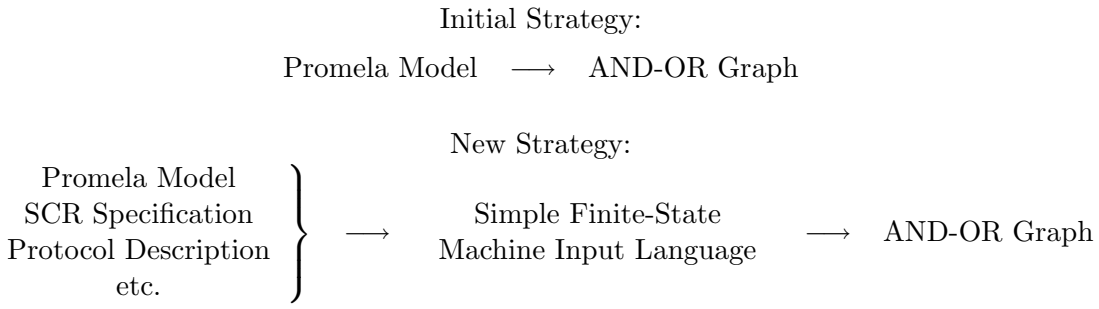


Figure 3.1: Example model from [27] as Promela (left) and a hand-translated AND-OR graph (right).

The work presented in this thesis began with an attempt to create a program that would automatically translate from Promela models like the one on the left of Figure 3.1 to AND-OR graphs like the one on the right. The first translator was written for the UNIX utility AWK, a slow but very convenient language for parsing the Promela input.

Promela represents an extended form of the finite-state machine model, with a variety of features including a special way of handling byte and integer variables and the *channel* data type. As the AWK translator program was built up to include more of the idiosyncrasies of Promela, it became very large and complex, so the decision was made to start over with a new strategy. Because Promela and other model checker input languages are based on finite-state machines (and because SPIN is able to automatically translate from Promela to finite-state machines), we decided to create our own very simple but formally specific finite-state machine input language, and translate from models written in that language into AND-OR graphs. Later on we would be able to create modules to translate from Promela, SCR, protocol specifications, etc. to our finite-state model language, and then to AND-OR graphs:



3.1.2 Translation

The finite-state machine model we used as a basis for AND-OR graph translation is slightly different from the more general definition of finite-state machines on page 12:

- Each finite-state machine $M \in S$ is a 3-tuple (Q, Σ, δ).
- Q is a finite set of states.
- Σ is a finite set of input/output symbols.
- $\delta : Q \times B \longrightarrow Q \times B$, where B is a set of zero or more symbols from Σ, is the transition function.

To show the input language we used to represent these models, here we translate by hand from the Promela example above into our input language. In addition to verifying and simulating the execution of Promela models, the model checker SPIN can output finite-state machine transition functions. Figure 3.2 shows SPIN's finite-state machine transition functions for the Promela model from Figure 3.1. There is a transition function listed for each of the processes defined in the original Promela model (*proctype A*, *proctype B*). Each line below the process name represents one transition. For example, if *proctype A* is in state 11, it can transition to state 9, but only if f is equal to 1. Or if *proctype A* is in state 3, it can transition to state 11, and in doing so as a side effect set a equal to 2.

In this simple example the only mathematical operators used on variables are "=" (assignment) and "==" (boolean equality test). The model would be much more complex if, for example, some line of Promela used the "++" (increment) operator; in that case the new state of the incremented variable would depend on its previous state, and therefore a transition would have to be defined for each of the possible previous states. But in this case operators are used in such a way that each line in Figure 3.2 represents only one transition. In terms of the formal description above:

```
proctype A
     state 11 -> state  9 => (f == 1)
     state  9 -> state  3 => (a == 1)
     state  9 -> state  5 => (a == 2)
     state  9 -> state  7 => (a == 3)
     state  3 -> state 11 => a = 2
     state  5 -> state 11 => a = 3
     state  7 -> state  8 => f = 0
     state  8 -> state 11 => a = 1
proctype B
     state 11 -> state  9 => (f == 0)
     state  9 -> state  3 => (b == 1)
     state  9 -> state  5 => (b == 2)
     state  9 -> state  7 => (b == 3)
     state  3 -> state 11 => b = 2
     state  5 -> state 11 => b = 3
     state  7 -> state  8 => f = 1
     state  8 -> state 11 => b = 1
```

	From State	Input	Output	To State
	11	f == 1		9
δ_A	9	a == 1		3
	9	a == 2		5
	9	a == 3		7
	3		a = 2	11
	5		a = 3	11
	7		f = 0	8
	8		a = 1	11
	11	f == 0		9
δ_B	9	b == 1		3
	9	b == 2		5
	9	b == 3		7
	3		b = 2	11
	5		b = 3	11
	7		f = 1	8
	8		b = 1	11

Figure 3.2: Finite-state machines for Figure 3.1 as output by SPIN (left) and as a typical transition function chart (right).

- $S = \{M_A, M_B\}$.
- $M_A = (Q_A, \Sigma_A, \delta_A)$.
- $Q_A = \{11, 9, 3, 5, 7, 8\}$.
- $\Sigma_A = \{(\texttt{f == 1}), (\texttt{a == 1}), \ldots (\texttt{a = 2}), (\texttt{a = 3}), \ldots\}$, etc.

Before describing the specifics of the finite-state machine input language and automatic translation procedure, we need to go back to the hand-translated AND-OR graph shown in Figure 3.1. In that AND-OR graph, AND-nodes are labeled "and," and all other nodes are OR-nodes. To believe an AND-node is true, we must believe all of its parents are true (the AND represents the conjunction of its parents); to believe an OR-node is true, we need only believe one of its parents (the OR represents the disjunction of its parents). There is no difference between solid and dotted edges; the graph was drawn that way to make it easier to read. One important difference between the AND-OR graph and the Promela code is that in the AND-OR graph there are no "==" (equality test) operators. These are unnecessary because the incoming edges to an "=" node represent assignment, and the outgoing edges represent "==."

The first AWK translation program produced AND-OR graphs like the one in Figure 3.1, with only AND-nodes, OR-nodes, and directed edges. But there is a problem with this representation: what if we concluded that $a = 1$ and $a = 2$—at the same time! Obviously

```
begin A
11;
9;
3;
5;
7;
8;
11; f=1; -;     9;
9;  a=1; -;     3;
9;  a=2; -;     5;
9;  a=3; -;     7;
3;  -;     a=2; 11;
5;  -;     a=3; 11;
7;  -;     f=0; 8;
8;  -;     a=1; 11;
end A
```

```
begin a
a=1;
a=2;
a=3;
end a

begin f
f=0;
f=1;
end f
```

```
begin A
11;
9;
11; f=1; -;       9;
9;  a=1; a=2;     11;
9;  a=2; a=3;     11;
9;  a=3; a=1,f=0; 11;
end A
```

```
begin a
a=1;
a=2;
a=3;
end a

begin f
f=0;
f=1;
end f
```

Figure 3.3: *Proctype A* and variables a, f from Figure 3.2 as input for NAYO translator (left) and in a simpler but equivalent form (right).

this is a contradiction, but the graph has no way of telling us (we know only because of assumptions about the node names). So we add a new feature to the graph: NO-edges. A NO-edge is an undirected edge between two nodes indicating that they are mutually exclusive. The normal directed edge becomes a YES-edge, which gives us the *NAYO* graph of Menzies et.al. [29, 31]:

- A set Y of directed YES-edges.
- A set O of OR-nodes—an OR-node is TRUE if any of its YES-edge parents are TRUE.
- A set A of AND-nodes—an AND-node is TRUE if all of its YES-edge parents are TRUE.
- A set N of undirected NO-edges connecting incompatible nodes.

The finite-state model for *proctype A* (from Figure 3.2), written in our finite-state machine input language, is shown in Figure 3.3. Here each finite-state machine description begins with a list of its states (this has to do with the creation of the right set of NO-edges, which will be explained in the next section); then transitions are listed (current state, input, output, next state). Global variables (a) and (f) are represented by machines without any transitions, but with states accessible to other machines. Note that the "==" operator is not present. Transitions with an "=" node in the input column represent the equality test, and transitions with an "=" node in the output column represent assignments.

```
 1: for (each finite-state machine) do
 2:   for (each state declared) do
 3:     Make an OR-node.
 4:     Connect it to the rest of this machine's OR-nodes with NO-edges.
 5:   end for
 6:   for (each transition in this finite-state machine) do
 7:     if (input(s) specified) then
 8:       Make an AND-node.
 9:       Make the current state a YES-edge parent of the AND-node.
10:       Make the input(s) (a) YES-edge parent(s) of the AND-node.
11:       Make the next state a YES-edge child of the AND-node.
12:       Make any output(s) (a) YES-edge child(ren) of the AND-node.
13:     end if
14:     if (no input specified) then
15:       Make the next state a YES-edge child of the current state.
16:       Make any output(s) (a) YES-edge child(ren) of the current state.
17:     end if
18:   end for
19: end for
```

Figure 3.4: Automatic translation procedure—finite-state machine(s) $\longrightarrow$ NAYO graph.

3.2 Automatic Translation and Properties of NAYO Graphs

3.2.1 Automatic Translation Procedure

To translate from the finite-state machines of Figure 3.3 to a NAYO graph, we use the procedure shown in Figure 3.4. Each state machine defined in the input begins with a list of its states. In line 3 of Figure 3.4, an OR-node is created for each state. An OR-node is used because there may be more than one way to reach this state—we consider the state reached if it is reached in *any* of these ways. Next, NO-edges are drawn connecting the new OR-node to the OR-nodes representing other states within the same finite-state machine. A finite-state machine can only be in one state at a time, so the OR-nodes representing its states are mutually exclusive.

After the list of states, each state machine definition continues with a list of transitions. Line 6 of Figure 3.4 begins the loop run on each transition. For each transition, there are two cases: (1) a set of input conditions is required to trigger the transition, or (2) no input is required; that is, the transition can nondeterministically go forward at any time. In the first case, we go to line 8 and make an AND-node for the transition; then we make the

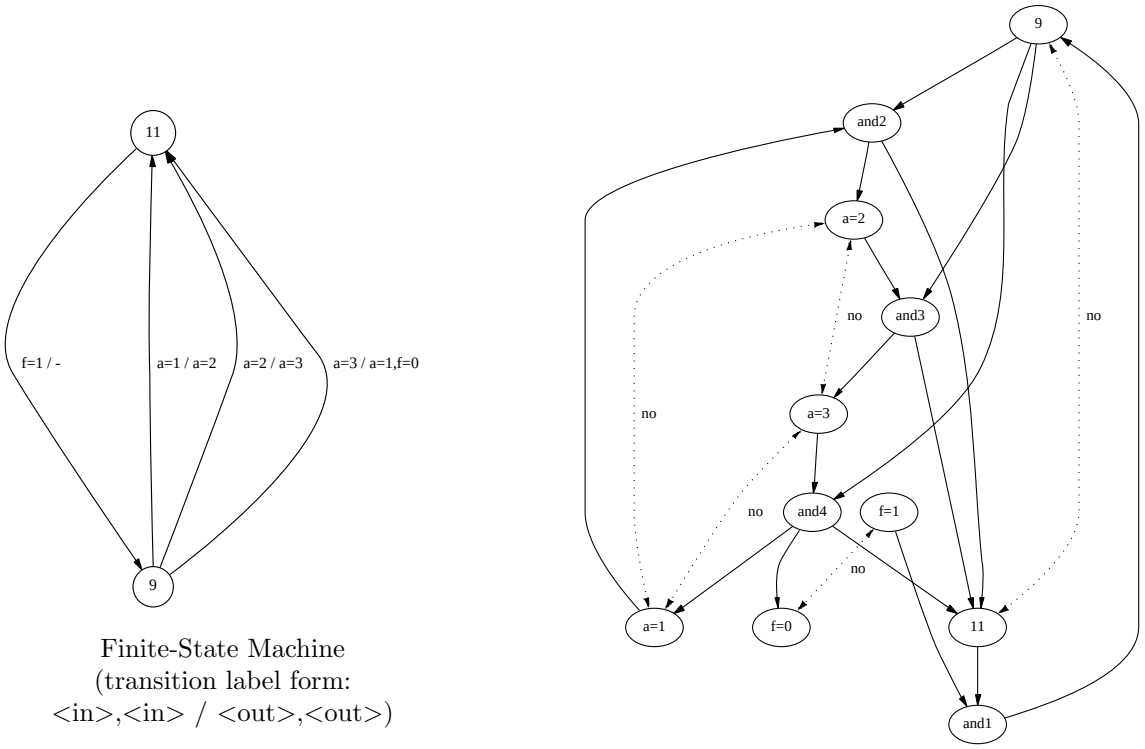


Figure 3.5: Finite-state model from Figure 3.3 as NAYO Graph.

current state OR-node (column 1 of the transition definition) and any *input* nodes parents of the new AND-node. The *input* nodes (column 2 of the transition definition) will be OR-nodes representing states in other machines. The AND-node represents the conjunction of everything required for the transition to go forward, so we make it the parent of the *next state* OR-node (column 4 of the transition definition). If any *output* is listed listed (column 3 of the transition definition), we also make these *output* nodes children of the AND-node. Like the *input* nodes, the *output* nodes will also be OR-nodes representing states in other state machines. The *output* nodes can be thought of as side effects of the transition.

For a transition with no input listed (case 2 above), which represents the situation in which the machine can nondeterministically move forward at any time, there is no need to create an AND-node. This is because the only precondition to the transition is the *current state*; that is, the current state represents everything required for the transition to go forward (like the AND-node in case 1). So we make it the parent of the *next state*, and if there are any outputs listed make these children of the *current state* as well.

1: **for** (each variable x_i) **do**
2: Make an OR-node for each literal $(x_i, \bar{x}_i)$.
3: Connect the two literals with a NO-edge.
4: **end for**
5: **for** (each clause) **do**
6: Make an OR-node.
7: Make this clause's 3 literals parents of the OR-node.
8: **end for**
9: Make an AND-node.
10: Make every clause's OR-node a parent of the AND-node.

Figure 3.6: Procedure to transform a 3SAT query into a NAYO graph.

In the finite-state machines shown on the left side of Figure 3.3, transitions are separated into no-input and no-output groups, because that's the way they are output by SPIN (Figure 3.2). But for our purposes the separation is unnecessary, so we can simplify the model to the one on the right, which will make the NAYO graph produced by the translator a readable size. Applying the translation procedure to the finite-state machine input example shown on the right side of Figure 3.3, we get the NAYO graph shown in Figure 3.5.

3.2.2 Properties of Resulting NAYO Graphs

In general, for a system of k finite-state machines with n states and m single-input, single-output transitions per machine, the resulting NAYO has:

- mk AND-nodes + nk OR-nodes = $O((m+n)k)$ nodes.
- $4mk$ YES-edges + $(n/2)(n-1)k$ NO-edges = $O((m+n^2)k)$ edges.

A finite-state machine composite (which would be exhaustively searched by a model checker) for the same system will in the worst case require $O(n^k)$ states and $O(n^{k-2})$ transitions [13].

Unfortunately it is not easy to find consistent assignments in a NAYO graph (if there are NO-edges present). In fact the problem of determining whether a particular node can be reached is NP-complete,[1] which we will show here in two steps.

- 3SAT $\leq_P$ NAYO search (NAYO search is at least as hard as the 3SAT problem, which is known to be NP-complete):

[1] *NP* is the class of problems for which a solution can be verified in polynomial time (the time required is a polynomial function of the input size); an *NP-complete* problem is (1) at least as hard as all problems in the class NP and is (2) itself in NP.

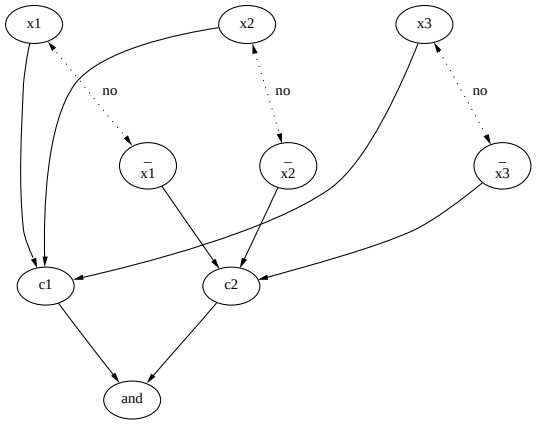


Figure 3.7: NAYO graph representing the 3SAT query $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x_1} \vee \bar{x_2} \vee \bar{x_3})$.

For the 3SAT problem we have a Boolean expression that is the conjunction of a series of clauses, each of which is the disjunction of 3 literals. A literal is either a variable (x_i, for example) or its negation ($\bar{x_i}$). The problem is to determine whether the expression is *satisfiable*; that is, does there exist an assignment of values to the variables that satisfies the total conjunction?

Figure 3.6 shows a simple procedure for transforming a 3SAT query into a NAYO graph. A NAYO graph created by this procedure will have a single AND-node; if this AND-node can be reached then the original 3SAT query is satisfiable. Figure 3.7 shows a NAYO graph representing a very simple 3SAT query.

- NAYO Search $\in$ NP:

  Clearly we can verify a NAYO search solution in polynomial time; we would (1) verify that the solution is a valid path of YES-edges, which requires $O(n-1)$ time (where n is the number of nodes in the NAYO graph); (2) verify that no two nodes in the solution path are connected by a NO-edge, which requires $O(n(n-1))$ time.

There is a tradeoff here: with a model checker, we can quickly search an exponentially large composite finite-state machine; with our NAYO graph technique we can represent the information in a small space but require exponential time to exhaustively search it. The focus of model checking research has been to reduce the number of states required by the finite-state composite—to solve the *state space explosion* problem. Our NAYO graph scheme solves the state space explosion, but creates a new problem, since exhaustive search requires

exponential time. The solution to this problem is to do a partial (not exhaustive) random search (see page 6).

Note: although it is easy to transform a SAT query into a NAYO graph, in general NAYO graphs have a much less organized structure than SAT queries. For example, a NAYO graph may contain many AND-nodes that are parents of each other; also, NAYO graphs do not just have contradictory pairs (like the Boolean variables in a SAT query) but may have an arbitrary number of nodes that contradict each other. So we have not attempted to translate NAYO graphs into SAT queries; if this could be done, it would allow us to take advantage of a huge amount of work done on SAT problems—we could use SAT algorithms to solve NAYO search. But here we have chosen a relatively simple random search scheme, instead of pursuing the task of translating NAYO graphs to SAT queries, which would likely be very difficult.

3.3 Simple Random Search of NAYO Graphs

Model checking techniques are used to exhaustively search a system of communicating finite-state machines (see page 10). They do this by building a composite finite-state machine, which represents all possible execution interleavings of the individual machines in the input. A state in the composite representing the machines shown in Figure 3.3, for example, might be $((A = 9), (a = 0), (f = 1))$. This state in the composite represents the situation in which the individual machines are in the states listed. This, again, is why the *state space explosion* occurs: the size of the composite grows exponentially large, because the number of composite states needed is equal to the product of the number of states in all of the individual machines.

Our goal is to find consistent assignments for the OR-nodes representing states in the individual finite-state machines in the input (in the language of abduction, these would be *worlds*—see page 6). These *consistent assignments* are the states in the composite finite-state machine that would be searched by a model checking tool. The key point here is that we want to find the composite states without actually building the composite—to avoid the state space explosion. As we noted earlier however, because of the NO-edges present in a NAYO graph, an exhaustive search for consistent assignments requires exponential time. So we instead propose a partial random search.

Our partial random search is desinged to solve the following problem: given some (not necessarily consistent) input set of OR-nodes, find an output set consistent with at least part

```
1: struct node
2:    yesList (list of children via YES-edges)
3:    noList (list of children via NO-edges)
4:    type {AND,OR}
5:    disqualified {TRUE,FALSE} (initially FALSE)
6:    waitValue (integer ≥ 0)
7:    found {TRUE,FALSE} (initially FALSE)

          These lines initialize the graph:

8:  for (∀ n) do
9:    if (n.type = OR) then
10:     if (n ∈ input) then
11:       n.waitValue ← 0.
12:       Put n into queue at random index.
13:     else
14:       n.waitValue ← 1.
15:     end if
16:   else if (n.type = AND) then
17:     n.waitValue ← | parents of n |.
18:   end if
19: end for

          The main search procedure:

20: while (queue ≠ ∅) do
21:   n ← pop(queue).
22:   if (n.disqualified = FALSE) then
23:     for (∀ n′ ∈ n.noList) do
24:       n′.disqualified ← TRUE.
25:     end for
26:     for (∀ n′ ∈ n.yesList) do
27:       if (n′.waitValue > 0) then
28:         n′.waitValue ← n′.waitValue − 1.
29:         if (n′.waitValue = 0) then
30:           n′.found ← TRUE.
31:           Put n′ into queue at random index.
32:         end if
33:       end if
34:     end for
35:     n.disqualified ← TRUE.
36:   end if
37: end while
```

Figure 3.8: Random search procedure for NAYO graphs (when search finishes, input nodes and nodes with `found = TRUE` have been reached).

of the input, and make that output set as large as possible. Ideally the output set contains an OR-node for a state in each of the finite-state machines from the original model—if so, the output is equivalent to one of the states in the composite finite-state machine that would be searched by a model checker. But in general, because the random search is not exhaustive, it may not tell us quite as much as a more time- (or space-) consuming technique; that is, the output set will constitute a *partial* description of a state in the composite.

The goal of our random NAYO graph search is, again, to find a consistent assignment of OR-nodes or, in terms of the equivalent composite finite-state machine, a partial description of a composite state. Figure 3.8 shows the random search in detail. Each node is stored with fields containing a list of its children via YES-edges, a list of its children via NO-edges, and its type (lines 1–3). In line 5 there is a field called *disqualified.* If a node is reached via a NO-edge, it is disqualified from the search, because it contradicts some node already reached. The *waitValue* field in line 6 is used to determine when a node is reached. A node's *waitValue*= 0 when it is reached (we are done waiting for the node to be reached). Each time a node is found to be the YES-edge child of some other reached node, its *waitValue* is decremented. An OR-node begins with *waitValue*= 1, so that if one parent is reached, the OR-node's *waitValue* will be decremented to zero—the OR-node will be reached. An AND-node begins with a *waitValue* equal to the number of parents it has, and each time one of its parents is reached, its *waitValue* is decremented. If all of its parents are eventually reached, its *waitValue* will have by that time been decremented all the way down to zero—it will be reached. The *found* field in line 7 is set to true for a node when that node's *waitValue* changes from 1 to 0. This is a way of marking nodes that are reached but were not included in the input set.

In lines 8–19 of Figure 3.8, the NAYO graph's nodes are set up so that they all have the correct *waitValue.* A set of (at least 1 of the) OR-nodes needs to be designated ahead of time as *input.* These are the nodes we believe to be true before the search starts. It's okay if there are mutually exclusive nodes in the input set; the search will throw out one or the other of any mutually exclusive pair. There are no AND-nodes in the input set. This is because the translation procedure from finite-state machines (Figure 3.4) creates only OR-nodes for meaningful states in the input (AND-nodes just link these states together). To begin the search, all of the input OR-nodes are put into a priority queue in random order (line 12).

Lines 20–37 form the main search procedure. At the beginning of each iteration of the loop, as long as there is at least one node in the queue, the first node is removed from the queue and checked to see if it has already been disqualified from the search. If it's not

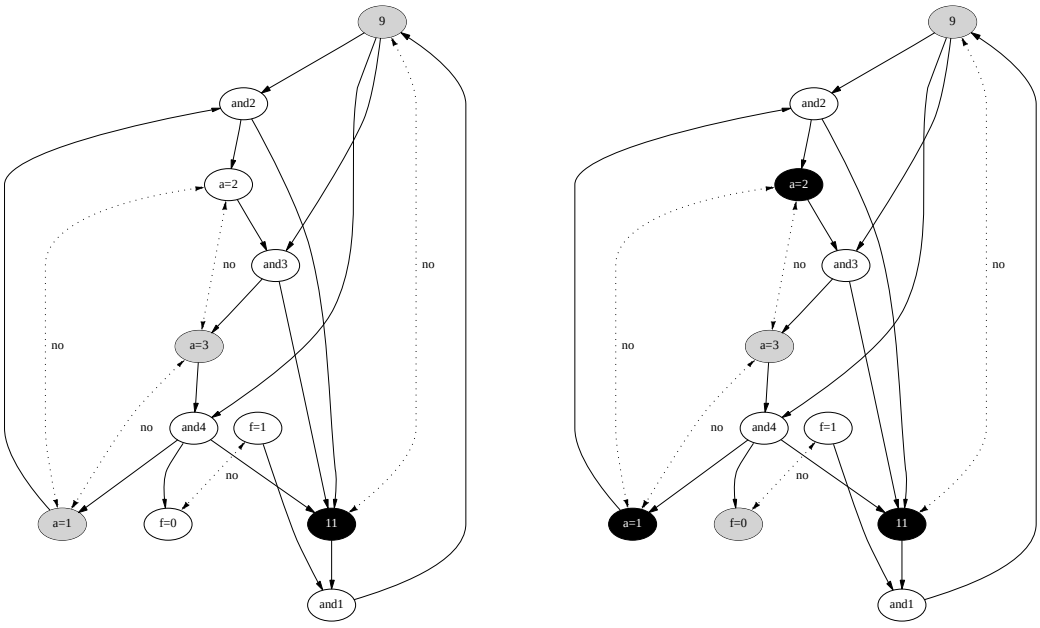


Figure 3.9: Random search progress through the NAYO graph shown initially in Figure 3.5 (gray indicates an OR-node with *waitValue*= 0; black indicates a disqualified OR-node).

disqualified, we consider it reached, and disqualify all nodes that contradict it (line 24). Then, for each of its children via YES-edges, if the child node is not disqualified and has not yet been reached (its *waitValue*> 0), we decrement the child node's *waitValue* (line 28). If any of the child nodes' *waitValue* becomes 0 at this point, we consider that node reached and put it into the queue at some random index. Line 35 disqualifies the node most recently removed from the queue so that if it is reached again (via a cycle) it will be ignored.

To give an example, we will follow the search through the NAYO graph shown in Figure 3.5. Suppose we start out with input OR-nodes 9, $a = 3$, and $a = 1$ in the queue (in that order). All the other OR-nodes will have an initial *waitValue* of 1; all the AND-nodes will have an initial *waitValue* of 2, since they all have 2 parents. We begin the search (the part beginning in line 20 of Figure 3.8) by removing 9 from the queue. Since there is a NO-edge between 9 and 11, we disqualify 11. We then decrement the *waitValue* for all of the children of 9—that is, *and2, and3,* and *and4,* whose *waitValues* are now all 1. The NAYO graph on the left side of Figure 3.9 represents the present situation: 9, $a = 3$, and $a = 1$ have *waitValue*= 0, and 11 is disqualified.

$a = 3$ and $a = 1$ are still in the queue, so the second iteration of the while loop begins with the removal of $a = 3$. $a = 1$ and $a = 2$ are disqualfied since they are not compatible with $a = 3$, and the *waitValue* of *and4* is decremented, since it's a child of $a = 3$. The

waitValue of *and4* is now 0, so we put it into the queue at some random index. Suppose it goes in first, ahead of $a = 1$. We begin the next iteration of the while loop by removing it, and decrementing its child, $f = 0$. At this point we have the NAYO shown on the right side of Figure 3.9. We then put $f = 0$ into the queue at some random index, and in progressive iterations remove it and $a = 1$, neither of which will make it possible to reach any new nodes, so the search ends with the NAYO pictured on the right side of Figure 3.9 (note: because the search makes random choices between contradictory options, this example result is just one of several possible results).

Chapter 4

Random Search of Example Models

In this chapter, the simple random search of Section 3.3 is modified to accommodate features of several example models. First, the search is extended to move across time (4.1), finding a series of internally consistent sets representing successive states of the program in execution. This version of the search is illustrated with an example model originally written in the SCR (software cost reduction) requirements specification language, the *Space Shuttle Liquid Hydrogen Subsystem* model from Atanacio [2]. Next the search and finite-state machine input language are modified to handle models of communications protocols, in which transitions may be triggered by messages passed from one finite-state machine to another (4.2). These messages are different from shared global variables: once a message is used to trigger a transition, it is *consumed* and therefore no longer available to trigger another transition. The message passing capability is illustrated with models of the alternating bit protocol and TCP (transport control protocol). In section (4.3), we track whether a specific node representing a property has been reached. We then use the random search to verify a simple safety property for a model of Dekker's solution to the two-process mutual exclusion problem (the example comes from Holzmann [10]).

4.1 Random Search Across Time

4.1.1 Modified Random Search Procedure

The random search procedure shown in Figure 3.8 inputs a (not necessarily consistent) set and outputs a consistent set, which is itself consistent with at least part of the input; that is, the procedure finds one partially defined state from the composite finite-state machine that would be searched by a model checking tool. But the verification done by model checking tools simulates *execution* of the program represented by the model: instead of just looking

at one of the composite's states, model checkers look at *all* of the composite's states, and all of the sequences of those states possible in the execution of the program represented by the model.

In order to extract meaningful information about program execution, a search cannot just look at individual states, but must look at sequences. So we need to modify the random search procedure from Figure 3.8. Figure 4.1 shows the modified version. Instead of producing a single output set, we want to produce a series of sets—a series of partial states from the composite (or partial *worlds,* in abduction terminology)—in order of some possible execution of the program represented by the model. First of all, the composite finite-state machine is like any other finite-state machine: its states are mutually exclusive. This means that in our series of sets (output by the modified random search), from one set to the next, there will be at least one NO-edge from a node in set i to a node in set $i+1$. We will therefore have at least one node that is disqualified in one set but not in the next. So that we don't have to keep resetting nodes' *disqualifed* field, the new search uses an integer for this field, and sets it equal to the time value (successive sets are identified by increasing time values) at which it was most recently disqualified (line 1). The *found* field in line 2 is modified here to keep track of not only which nodes have been reached, but at what time the node was most recently reached.

We intialize nodes' *waitValues* and put input nodes into the queue just as before (lines 8–19 of Figure 3.8). The main search procedure (lines 3–36) is a loop structure that iterates through increasing time values until the MAX time value is reached (line 4). In each iteration, a slightly modified version of the main loop from the old search (lines 5–23 here; lines 20–37 of Figure 3.8) is followed by routine that sets everything up for the next iteration (lines 24–35).

In our new version of the old procedure's main loop, the *disqualified* field is assigned or checked for an integer value, which indicates at what time a node has been disqualified (the value of *disqualified* would have been just *true* or *false* before). Also, the *found* field is updated according to the most recent time a node has been reached.

Lines 24–35 set us up for the next iteration. With each iteration, we want to move forward (in terms of the execution of the program represented by the model). Lines 26–27 select two groups of nodes to use as input for the next iteration. The first group is the set of nodes reached by the current iteration (these have been marked *found* and then disqualified in lines 20–21 to prevent cycling). We also take a *frontier* of nodes—nodes that are, in a sense, *almost* reached. The frontier is made up of nodes found in line 15 and already disqualified

Each node n like Figure 3.8, except:

1: `disqualified` (was {`TRUE`,`FALSE`}, now integer ≥ 0)
2: `found` (was {`TRUE`,`FALSE`}, now integer ≥ 0)

No changes to Figure 3.8 initialization procedure.

The new main search procedure (lines 5–23 are identical to the main search procedure in Figure 3.8, except 7, 9, 15, and 20–21):

```
3: time ← 0.
4: while (time ≤ MAX) do
5:   while (queue ≠ ∅) do
6:     n ← pop(queue).
7:     if (n.disqualified ≠ time) then
8:       for (∀ n′ ∈ n.noList) do
9:         n′.disqualified ← time.
10:      end for
11:      for (∀ n′ in n.yesList) do
12:        if (n′.waitValue > 0) then
13:          n′.waitValue ← n′.waitValue − 1.
14:          if (n′.waitValue = 0) then
15:            n′.found ← time.
16:            Put n′ into queue at random index.
17:          end if
18:        end if
19:      end for
20:      n.found ← time.
21:      n.disqualified ← time.
22:    end if
23:  end while
24:  for (∀ n) do
25:    if (n.type = OR) then
26:      if (n.found = time) then
27:        Put n into queue at random index.
28:      else
29:        n.waitValue ← 1.
30:      end if
31:    else
32:      n.waitValue ← | parents of n |.
33:    end if
34:  end for
35:  time ← time + 1.
36: end while
```

Figure 4.1: Modified random search procedure.

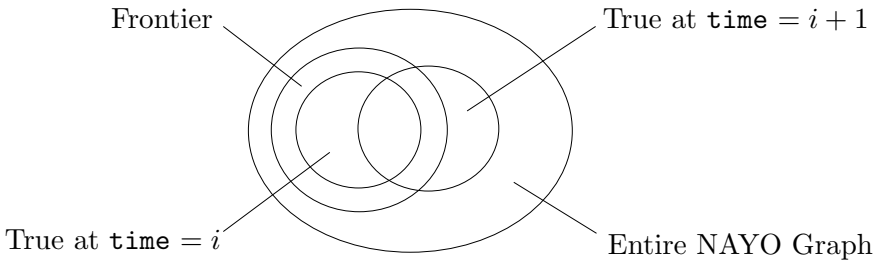


Figure 4.2: Sets involved in succesive iterations of the random search procedure shown in Figure 4.1.

for the current time because of a NO-edge. All other OR-nodes' *waitValue* fields are reset to 1 (line 29). In lines 31 and 32 we reset AND-nodes' *waitValues*, and in line 35 increment *time.* AND-nodes are never put into the queue as input for an upcoming iteration, because they do not represent meaningful states in the original finite-state model (see translation procedure shown in Figure 3.4).

Figure 4.2 shows a diagram of the sets involved in two successive search iterations. There is a set of OR-nodes known to be true at *time*= i, and there is a frontier of OR-nodes, nodes which at *time*= i are disqualified because of a NO-edge (nodes whose *disqualified* field = i). Both sets—the set true at *time*= 0 and the frontier set—are put into the queue and serve as input for the next iteration. The next iteration finds a set of OR-nodes true at *time*= 1, which will include some of the nodes in the two sets from the previous iteration (but not all, since there is at least one NO-edge between the two sets from the previous iteration).

4.1.2 SCR Specification Example: Space Shuttle Liquid Hydrogen Subsystem

The Software Cost Reduction (SCR) language, used for writing software requirements specifications, is formally based on finite-state machines [4]. It is relatively easy to rewrite SCR specifications in the finite-state machine form of Figure 3.3, as long as the specification does not use variables with a large number of possible values. In practice, this means making some assumptions about key values, and creating a new variable that can take on only those key values (the *abstraction* strategy proposed by Clarke et.al. [7]—see page 10).

In a technical report comparing SCR-based model checking tools (incorporating SPIN) to the SMV model checker, Atanacio includes an SCR specification for the *Space Shuttle Liquid Hydrogen Subsystem* [2]. Figure 4.3 shows this specification as, first, a set of finite-

state machines (upper left), and then translated into a NAYO graph (bottom right). In creating these models it was not necessary to reduce the number of values taken by any of the variables; this had already been done when the original SCR specification was written, in order to decrease the state space (and therefore the amount of memory and time) required by the model checker SPIN, which was used in conjunction with SCR tools to verify the model.

While the finite-state machines and NAYO graph in Figure 4.3 are too detailed to read clearly on one page, it is possible to see a few interesting things about the structure of the model. The far left finite-state machine has five states, and all others have only two. Going back to our translation process (Figure 3.4), this means the NAYO graph will have comparatively few NO-edges. If we had a smaller number of finite-state machines, but each machine had many more states, the NAYO graph would have many more NO-edges (states within a machine form a complete graph of NO-edges, but there aren't any NO-edges between machines).

The NAYO graph (lower right) at first appears to be an inscrutable jumble, but a close look reveals something strange in the upper right corner. There are six nodes in three pairs, with each pair connected by a NO-edge (NO-edges are dotted). These isolated nodes represent three Boolean variables declared but never used in the remainder of the specification. So the NAYO graph representation, although we are using it because it makes our random search scheme possible, is also useful for visualizing communicating finite-state machine systems (as a control flow or dependency graph might also be useful, for example).

Figure 4.4 shows results for a series of random searches on the NAYO graph version of the Space Shutle Liquid Hydrogen Subsystem from Figure 4.3. The vertical axis shows the number of unique OR-nodes reached (not including the input), and the horizontal axis shows the total number of OR-nodes reached (often the random search will find repeats). There are 20 searches plotted in Figure 4.4: 5 with $MAX = 1$, 5 with $MAX = 2$, 5 with $MAX = 3$, and 5 with $MAX = 4$ (see Figure 4.1; MAX is the maximum number of time values allowed each time the inner loop is executed). The results show the basic shape we are looking for (see Figure 1.1 on page 2 in the introduction)—a quick rise to a plateau that then remains constant indefinitely. We will try to clarify exactly what *quick* means after a few more modifications to the random search. For now we observe in Figure 4.1 that the number of unique OR-nodes reached rose to a plateau of height 22 and stayed there, and this happened in all of the trials.

Why 22? A close look at the finite-state machine input file for the SCR specification (see

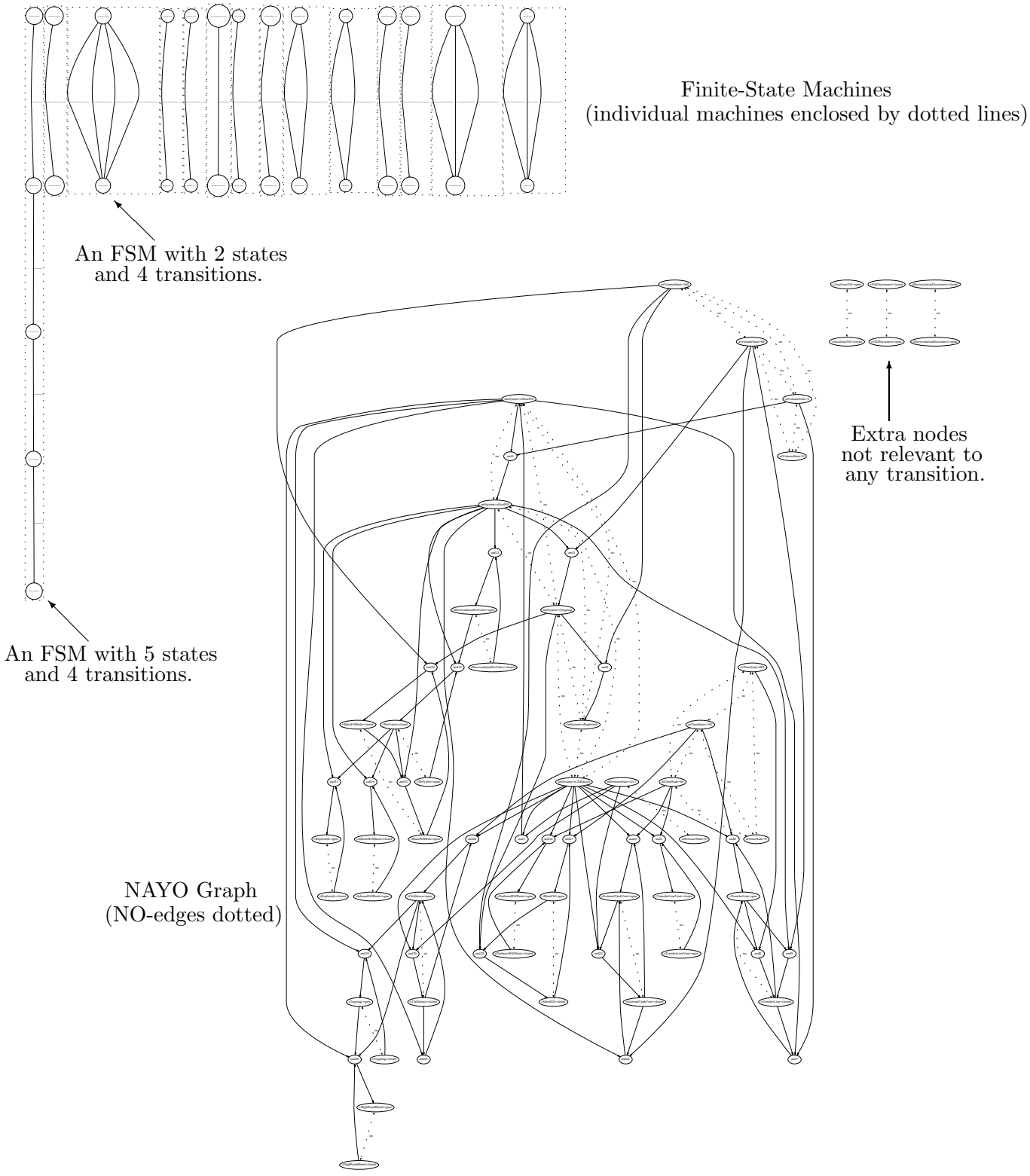


Figure 4.3: SCR requirements specification, the *Space Shuttle Liquid Hydrogen Subsystem,* as a set of finite-state machines (top left) and as a NAYO graph (bottom right).

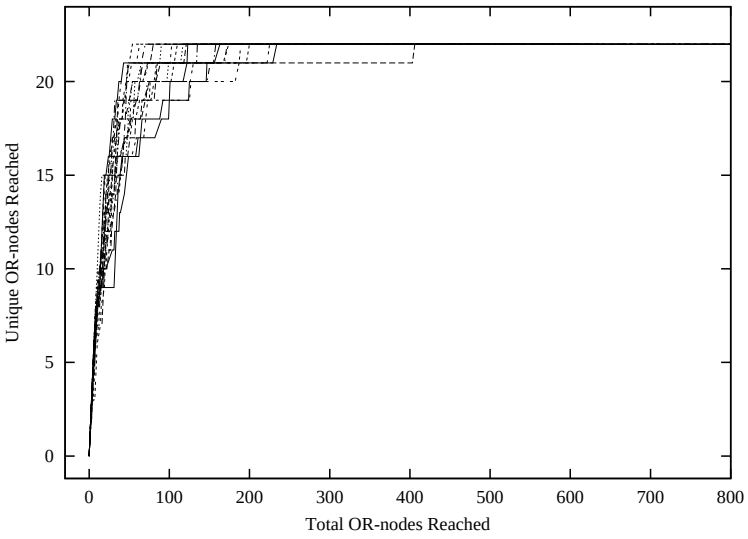


Figure 4.4: Result of random searches on SCR specification NAYO graph from Figure 4.3.

Appendix B.1) shows that there are exactly 22 unique *next state* entries in the transition charts for the individual finite-state machines. So for this model, which is small enough to read and understand directly, the random search finds everything you would expect.

4.2 Translation and Search Modified for Message-Passing Finite-State Models

4.2.1 Modified Translation and Search (alternating bit protocol example)

Figure 4.5 shows a finite-state machine and a NAYO graph representing the *sender* side of a simple version of the alternating bit protocol (see Figure 2.3, page 13, and Appendix B.2; this model comes from [13]). To simplify the Figure, the *receiver* side is not shown (the receiver is included in the transition function for this model [13] shown in Appendix B.2 and in the model we used for our experiments). There is an important difference between this example and the previous finite-state machines we have considered. Until now all transitions have been triggered by states in other machines, e.g., if machine x is in state 1 and machine y is in state 2, machine x may transition to state 2. But in this example we consider machines representing a communication protocol. A transition in the sender is not triggered by a state in the receiver, but rather a *message* from the receiver.

In order to model message passing between finite-state machines, we need to modify

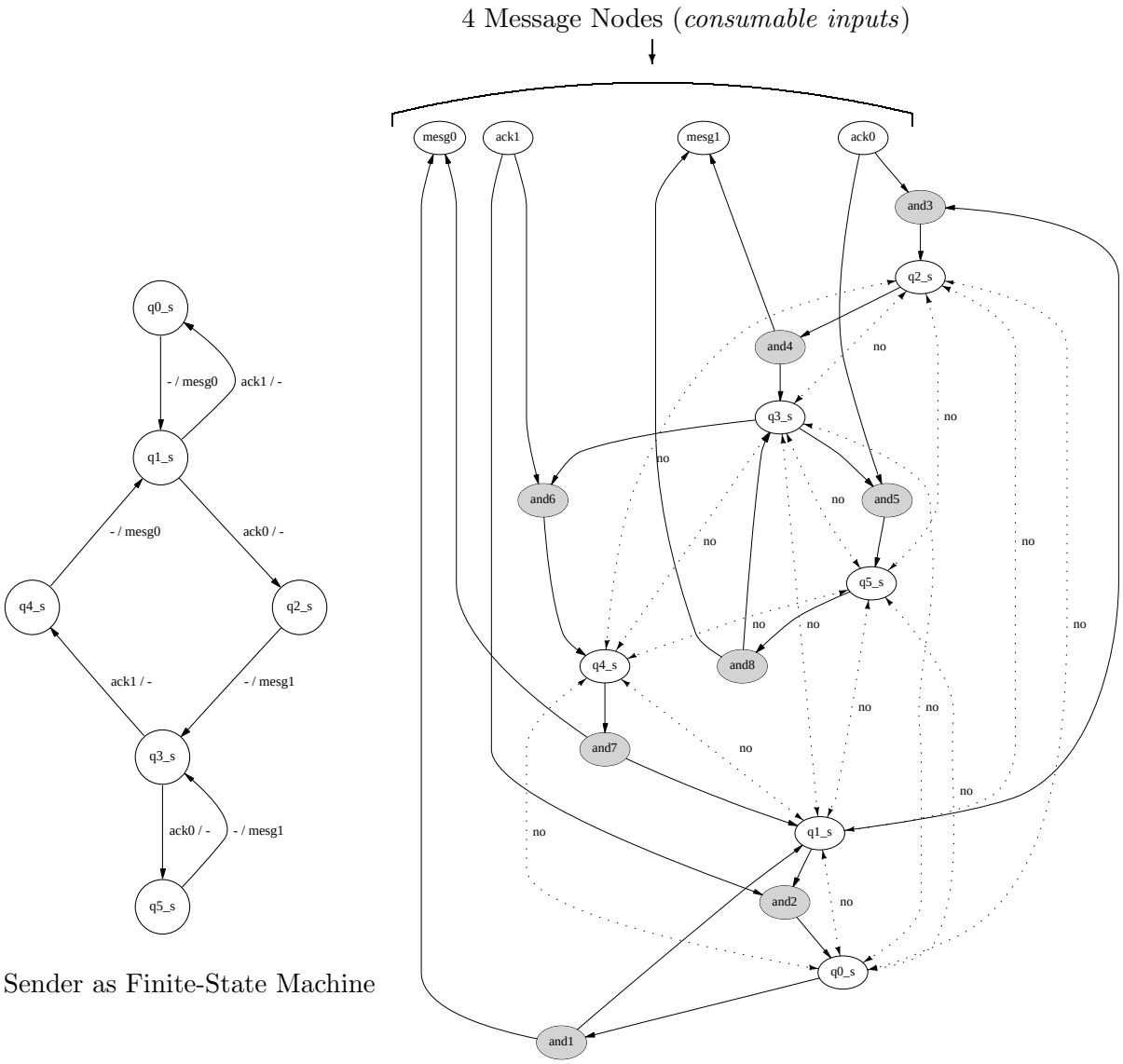


Figure 4.5: Finite-state model and NAYO graph for alternating bit protocol *sender* (*receiver* not shown).

Add to main search procedure after line 21:

```
1: if (n.type = OR) then
2:   n.waitValue ← 1.
3: end if
```

Figure 4.6: Section added to the random search procedure shown in Figure 4.1.

```
begin old
current state;  state input;     state output;     next state;
end old
```

```
begin new
current state;  state input,     state input,      next state;
                 message input;   state output,
                                  message output;
end new
```

Figure 4.7: Old (above) and new (below) forms for finite-state machine input.

the input language and random search procedure slightly. This is because a transition triggered by a message *consumes* the message. In previous examples, when transitions were triggered by states in other machines, the transition occurred without affecting the state of the other machine—the state was available as input to an arbitrary number of transitions. But messages are *consumable inputs* good for only one transition. At the top of the NAYO graph in Figure 4.5, the four message nodes are marked. If the second node of the four, *ack1,* became true, it could contribute to either *and6* or *and7,* but not both. Once the message was consumed by one of the AND-nodes it would no longer be available to the other.

To account for this behavior, the random search procedure needs just one minor change. When a node is removed from the queue, its children via NO-edges are disqualified (line 9 of Figure 4.1), the *waitValues* of its children via YES-edges are decremented (line 13), and if any of the YES-edge children are reached, they are put into the queue. At this point, if the node is an OR-node, the input it represents must be *consumed.* This is accomplished by resetting its *waitValue* to 1—we add the lines in Figure 4.6 after line 21 of Figure 4.1.

After making this change to the random search procedure, we need to modify the finite-state machine input form slightly to account for it. Figure 4.7 shows the old style finite-state machine input, where transitions are triggered by states in other machines (*state input*) or may force a transition in another machine (*state output*). In the new form, also shown in Figure 4.7, received messages may function as input to a transition (*message input*), and

messages may be sent as a result of a transition (*message output*). Also, there is a trick here to retain *state input* transitions: they are consumed, just like messages, but then *regenerated* by the transition. That's why *state intput* appears again in the third column. After the transition is completed, it will still be available to trigger other transitions.

4.2.2 TCP (transport control protocol)

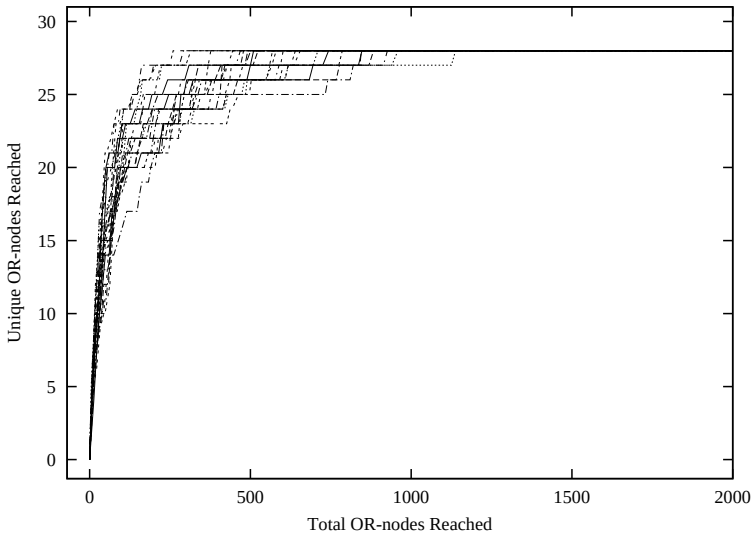


Figure 4.8: Result of random searches on TCP protocol model.

Figure 4.8 shows the result of a series of random searches on a NAYO graph representing the TCP finite-state machine diagram in Appendix B.3. For this example the new *consumable input* version of the random search was used, since state transitions in the TCP client or server finite-state machines are triggered by messages from, not states in, the other machine. The results plotted here look similar to those in Figure 4.4. As before a range of values were used for the *MAX* variable, and in all cases the search quickly rose to the same plateau value of 28 unique nodes reached. The number of total nodes reached in the process is approximately double that of the SCR example. One might wonder why all nodes were not reached—is there an error in the protocol or the model? Probably not. But the model has been written so that the messages triggering transitions come from the (external) applications actually using the protocol. We include these messages in the input for our search, but they are not counted as reached since it impossible to reach them without including them in the input.

This is an interesting result. From the point of view of our random search, the TCP example is about twice as hard to search as the SCR example; that is, approximately double the number of total OR-nodes reached is required to find approximately the same maximum number of unique OR-nodes reached. But if we were to construct the composite finite-state machine for each of these, the number of states required for the SCR example is bounded at about 10^7, while the number of states required by the TCP example is at most 121 (the bound on the number of transitions is about 10^7 for the SCR example and is 462 for the TCP example). If we were using an exhaustive model checking search scheme, we would consider the SCR example a much more difficult problem, because it involves many machines working in parallel and therefore requires a large number of states in the composite. The TCP example, on the other hand, involves only two finite-state machines, and requires a very small number of states in the composite.

Comparing the SCR and TCP examples, we start to see how the structure of the input model might favor the NAYO random search. For models with few processes running in parallel and many mutually exclusive states within each process, e.g., TCP, our NAYO random search must work much harder. But for models with many small parallel processes (by *small* we mean: few mutually exclusive states), like the SCR model of the Space Shuttle Liquid Hydrogen Subsystem, the NAYO search is likely to do very well.

4.3 Verifying Logical Properties with Random Search

The random search presented here, in addition to measuring testability for finite-state models, is able to test for simple logical properties. We believe it has the potential to be able to test for more complex temporal properties like those verified by model checkers, but here we will just show a simple example as motivation for further work.

Consider Dekker's solution to the two-process mutual exclusion problem, taken from [10] as the Promela code shown on the left side of Figure 4.9. The right side of Figure 4.9 shows SPIN's finite-state model interpretation of the Promela on the left. In *proctype A* and *proctype B* the lines marked *critical section* would be replaced with code to be performed without interference from the other process, i.e., the use of some shared resource. The basic *safety* property we will show for this example is that that A and B can never be simultaneously in their critical sections. In the finite-state version of the model, state 4 in each of the processes represents the critical section.

Figure 4.10 shows the model from Figure 4.9 in the finite-state model form ready to

```
#define true  1
#define false 0
#define Aturn false
#define Bturn true

bool x, y, t;

proctype A()
{
     x = true;
     t = Bturn;
     (y == false || t == Aturn);
     /* critical section */
     x = false
}

proctype B()
{
     y = true;
     t = Aturn;
     (x == false || t == Bturn);
     /* critical section */
     y = false
}

init
{
     run A();
     run B()
}
```

```
proctype A
  state 1 -> state 2 => x = 1
  state 2 -> state 3 => t = 1
  state 3 -> state 4 => ((y == 0) || (t == 0))
  state 4 -> state 5 => x = 0
  state 5 -> state 0 => -end-

proctype B
  state 1 -> state 2 => y = 1
  state 2 -> state 3 => t = 0
  state 3 -> state 4 => ((x == 0) || (t == 1))
  state 4 -> state 5 => y = 0
  state 5 -> state 0 => -end-

proctype init
  state 1 -> state 2 => (run A())
  state 2 -> state 3 => (run B())
  state 3 -> state 0 => -end-
```

For processes A and B, state 4 represents the area marked *critical section* in the Promela model.

Figure 4.9: Dekker's solution to the two-process mutual exclusion problem, as Promela from [10] (left) and as finite-state machines output by SPIN (right).

```
begin x
x;
!x;
end x

begin y
y;
!y;
end y

begin t
t;
!t;
end t
```

```
begin A
A1;
A2;
A3;
A4;
A5;
Aend;
A1; -;  x;  A2;
A2; -;  t;  A3;
A3; !y; -;  A4;
A3; !t; -;  A4;
A4; -;  !x; A5;
A5; -;  -;  Aend;
end A
```

```
begin B
B1;
B2;
B3;
B4;
B5;
Bend;
B1; -;  y;  B2;
B2; -;  !t; B3;
B3; !x; -;  B4;
B3; t;  -;  B4;
B4; -;  !y; B5;
B5; -;  -;  Bend;
end A
```

```
begin init
init1;
init2;
init3;
initend;
init1; -; A1; init2;
init2; -; B1; init3;
init3; -; -;  initend;
end init

begin safety
ok;
!ok;
ok; A4,B4; A4,B4; !ok;
end safety
```

Figure 4.10: Promela model from Figure 4.9 (originally from [10]) as finite-state machine input for NAYO translation and search.

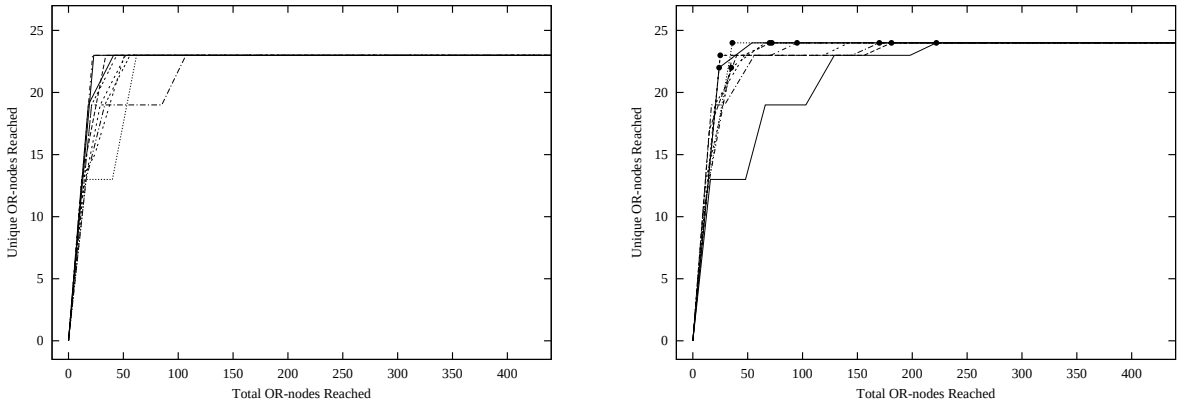


Figure 4.11: Search results for Dekker's mutual exclusion solution model (from Figure 4.9): original model (left), with error transition added (right).

translate into a NAYO graph. An extra finite-state machine has been added, called *safety,* which starts in state *ok* and transitions into state *!ok* if machine A is ever in state *A4* at the same time machine B is in state *B4*. The left side of Figure 4.11 shows random search results for the model. The searches quickly rise to a plateau without ever reaching *!ok*. So that the search can find a safety violation, we add the following error line to finite-state machine A:

```
A3; -; -; A4;
```

This line says that the machine may go directly from state *A3* to *A4*, the critical section state, without checking that either y or t is zero (zero is in this example equivalent to false). The right side of Figure 4.11 shows random search results for the model with the new error line. The black dots indicate at what point, for each search, a node violating the safety property *!ok* was found. The property was quickly found to be violated in every trial.

In this example we check for the same property at each iteration as the search progresses through time (can we reach *!ok* at *time*= 1? at *time*= 2?). To search for more complex temporal properties, we would check for components of the property at different times, and then check that the relationship between the different times is correct (for example, can we reach a at *time*= 1, b at some later time, and then eventually reach a again?).

Chapter 5

Confirming Example Results with Experiments on Randomly Generated Models

In chapter 5 we attempt to confirm that our random search on NAYO graphs will yeild the expected *plateau* result (see in chapter 4's several examples) for a wide range of NAYO graphs representing communicating finite-state machine models. We begin by (randomly) generating two example NAYO graphs representing finite-state models with structure similar to the SCR and TCP examples of chapter 4 and showing that the random search yields results for these graphs similar to the results of the search on the original SCR and TCP models (5.1). Next, we generate a large number of random NAYO graphs representing finite-state models, showing how the testability of many models can be compared (5.2). We then use Menzies and Hu's *TAR2* tool [31] to determine from our search data which finite-state model attributes are most significant to models' testability (5.3), and infer from these results some ideas about how more testable software might be designed.

5.1 Generating Random NAYO Graphs Representative of Finite-State Models

In the last chapter we compared the SCR Space Shuttle example NAYO search to the *consumable input* search of a NAYO graph representing the TCP protocol. We concluded that our NAYO random search scheme will work well for models like the SCR example, which has many small finite-state machines running in parallel, but will not work as well on models like TCP, which have fewer but larger finite-state machines. But these are just two isolated cases, and they are both such small models that the search could almost be done by

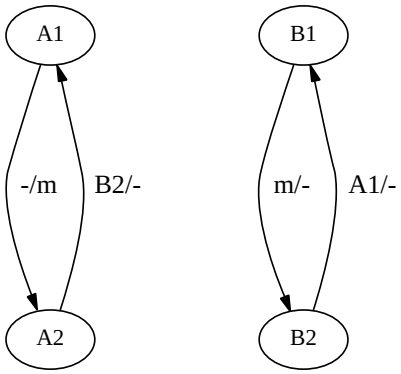


Figure 5.1: A model with (1) 2 finite-state machines, (2) 2 states per machine, (3) 2 transitions per machine, (4) 2 transition inputs that are states in another machine ("A1, B2"), (5) 1 consumable message ("m"), (6) 1 transition input that is a consumable message (the "m" on the right), and (7) 1 transition output that is a consumable message (the "m" on the left).

hand. We would like to compare a wide range of models, varying in many ways, including size, to see where the random NAYO search will work well and where it will fail.

While experiments have been done in the past on a wide range of NAYO graphs [25,27,29], we would like to focus on a subset of NAYO graphs: those that represent finite-state machines. Instead of randomly generating NAYO graphs directly, we generate communicating finite-state machine system models like the examples above, and then translate those into NAYO graphs. Our search results will be organized in terms of the finite-state machine input models rather than the NAYO graphs.

Past NAYO graph search experiments were motivated by the claim that reachability corresponds to a quantitative testability measure of the system represented by the model (this is discussed in more detail on page 6). Our goal in generating finite-state models and searching their representative NAYO graphs is to answer questions like: are systems with many machines more or less testable? How does the size of the individual machines affect systems' testability? What about the complexity of the transitions defined? The number of global variables? The variety of possible messages transmitted between machines?

The following parameters (illustrated in Figure 5.1) were used to generate finite-state models:

1. The number of individual finite-state machines in the system.
2. The number of states per finite-state machine.

3. The number of transitions per machine.
4. The number of inputs per transition that are states in other machines (the type of inputs used in the SCR Space Shuttle example).
5. The number of unique *consumable* messages that can be passed between machines.
6. The number of inputs per transition that are consumable messages (the type of inputs used in the TCP protocol example).
7. The number of outputs per transition that are consumable messages.

It may seem strange that there is no "maximum number of outputs per transition that are states ..." This is because the relationship between finite-state machines expressed by a transition with such an output is equivalent to the relationship expressed by an input that is a state in another machine. This is not true for message inputs and outputs, which must be generated by one transition in order to be consumed by another.

When generating the random finite-state models, certain rules were followed to keep them similar to real models. First, all machines have at least 2 mutually exclusive states (it wouldn't make sense to define a machine with 0 or only 1 state, although it would be possible to represent it in the NAYO graph). For transitions, the following rules were applied:

1. The *current state* and *next state* (columns 1 and 4 of the transition function chart) must come from the machine in which the transition is defined and must not match.
2. Inputs (in the second column of the transition function chart) that are states must come from *other* machines, and none may be mutually exclusive (the transition could never occur if it required mutually exclusive inputs).
3. The set of inputs that are messages from other machines contains no duplicates.
4. The set of outputs that are messages to other machines contains no duplicates.

Figure 5.2 shows the results of repeated random search on two randomly generated graphs. The first was a graph of approximately the same structure and size as the SCR Space Shuttle example, with many small machines and transitions triggered by states in other machines. The second graph was like the TCP protocol example, with a few large finite-state machines, and with transitions triggered by messages from other machines. Here again we see the *rise-to-plateau* output after several hundred total nodes reached. And again, in spite of the fact that the first random graph would have, as a composite finite-state machine, more than 10,000 times the number of states the second has, our results show that for the NAYO random search scheme, they are comparable in difficulty. In the next section

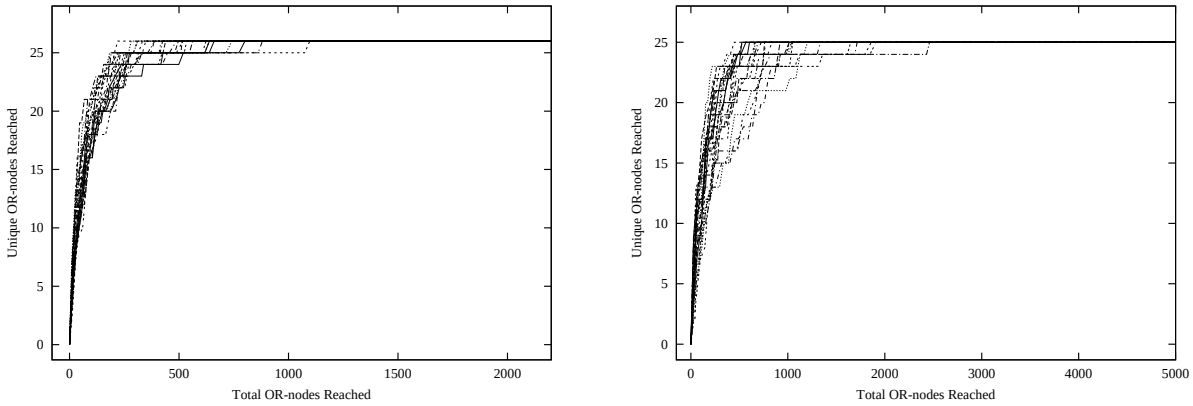


Figure 5.2: Result of random searches on random graph with attributes like those of the SCR specification (left) and TCP protocol model (right).

we will attempt to show in detail how the difficulty of searching (the testability) is related to the structure of the finite-state model.

5.2 Inferring Testability Measurements from Random Search

5.2.1 Search Results from a Wide Range of Models

In order to explore the relationship between finite-state model attributes and testability, we generated 15,000 NAYO graphs, according to the procedure described in the previous section, with the following attributes:

1. Between 2 and 20 individual finite-state machines.
2. Between 4 and 486 states (the sum of states in all individual machines, not the number of states that would be in the composite finite-state machine representing the entire system).
3. Between 0 and 272 transitions (the sum of transitions in all individual machines).
4. Between 0 and 737 transition inputs that are states from other machines (the sum for all transitions in all machines).
5. Between 0 and 20 consumable messages.
6. Between 0 and 647 transition inputs that are consumable messages.

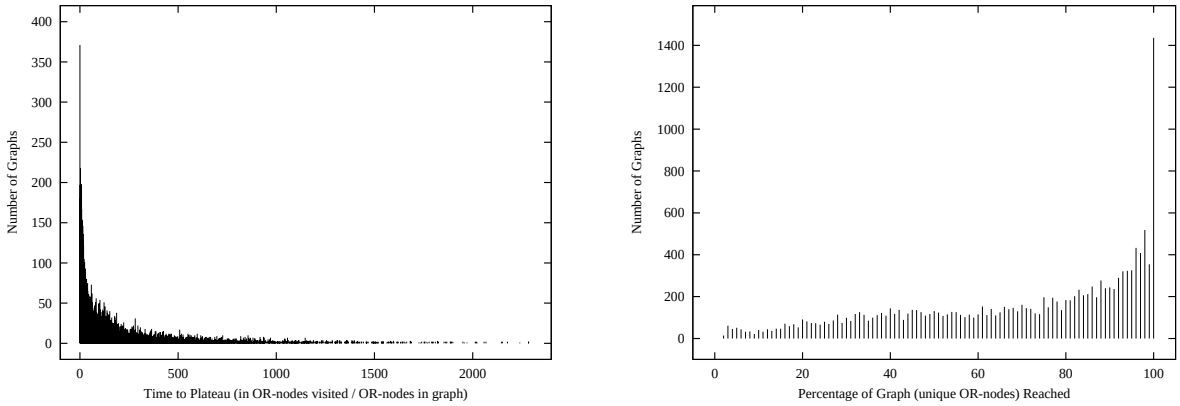


Figure 5.3: Summary of time-to-plateau (left) and plateau height (right) results for 15,000 NAYO graphs; average time-to-plateau = 1.376 × NAYO size, and average plateau height = 53.03%.

7. Between 0 and 719 transition outputs that are consumable messages.

1,000 individual random searches were done on each random graph, keeping track of the total number of OR-nodes visited and the number or unique OR-nodes reached. The result of this series of searches on a NAYO graph is a *plateau,* like the one shown in Figure 1.1 and those shown in Figures 4.4, 4.8, and 5.2. For each graph, the key search results are: (1) the height of the plateau (the percentage of the graph's OR-nodes reached by the random search), and (2) the time required to reach the plateau (*time* here is measured in the total number of OR-nodes visited by the search, divided by the number of OR-nodes in the graph).

We first consider the second of the key results listed above, the *time to plateau,* since it is the simpler one to interpret. The left side of Figure 5.3 shows that, for most graphs, a plateau is reached quickly. For about 12% of the NAYO graphs in this experiment, the plateau was reached after visiting a number of OR-nodes less than or equal to the number of OR-nodes in the graph; for about 39% the plateau was reached after OR-node visits less than or equal to 20 times the number of OR-nodes in the graph. In a few cases the time to plateau was nearly 2500 times the number of OR-nodes in the graph, but even this result is not bad in the proper context: exhaustive search alternatives work on a state space exponentially larger than the NAYO graph representation of the finite-state model. For example, there are 47 OR-nodes in the NAYO graph representing the Space Shuttle Liquid Hydrogen Subsystem model discussed on page 29. The upper bound on the number of states that would be required for the composite is about 10 million, which is much greater than

$2500 \times 47 = 117,500$.

Clearly, then, the random search reaches a plateau very quickly for a wide range of finite-state model inputs. This is what we expect, based on the *funnel theory* proposed by Menzies et.al. [31] and discussed on page 6: if it is true that a small number of key variables (a funnel) determines a large portion of program behavior, we expect to see that a small number of random searches will find the funnel and therefore find everything a large number of searches is capable of finding. But how much is it possible to find? Our working definition of testability is, very informally, that in a highly testable space we can find a lot quickly. The left side of Figure 5.3 shows that, however much we can find, we can find it quickly with the random search. So our definition of testability will be primarily concerned with how much it is possible to find—the height of the plateau.

The right side of Figure 5.3 shows a summary of plateau height (the number of unique OR-nodes reached as a percentage of the total number of OR-nodes in the graph) results for the same group of NAYO graphs used in the time-to-plateau experiment. Here we see a range of results with an encouraging spike at 100% percent (these are highly testable models), but still a significant number of graphs for which the random search found much less. We would like to know more about the attributes of finite-state models that are most testable.

5.3 Finite-State Model Attributes' Influence on Testability

In order to determine the relationship between finite-state machine attributes and testability (plateau height), we use Menzies and Hu's *TAR2* tool, a *treatment learner* [34]. The form of the data from our NAYO graph experiments lists various finite-state machine attributes' values, and then in the final column there is a number representing the class (or category) in which the result belongs. For example, the following line represents the result of 1,000 searches done on the NAYO graph version of a finite-state model with 48 communicating finite-state machines, 139 total states ... 141 OR-nodes in the NAYO graph, and *class-10* search results:

48, 139, 293, 2, 772, 589, 780, 141, 10.

In this case *class-10* just means that between 90% and 100% of the OR-nodes in the graph were reached. TAR2 summarizes a long list of lines like this by suggesting *treatments* that drive the data toward the highest class. A treatment is a restriction on one or more

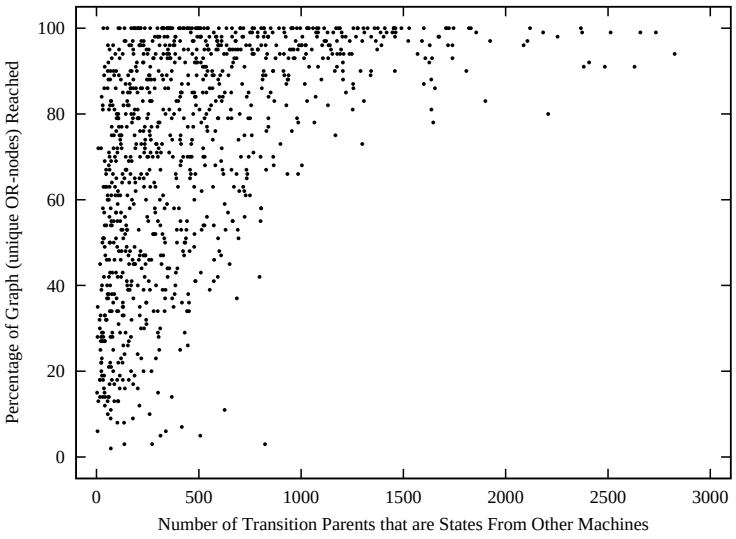


Figure 5.4: The relationship between the percentage of OR-nodes reached and the number of transition inputs that are states from other machines.

	⟵ Better			Worse ⟶		
machines	lowest	lowest	lowest	lowest	lowest	lowest
states	lowest	lowest	lowest	lowest	lowest	lowest
transitions	low	low	low	lowest	lowest	lowest
messages						
state inputs	high					lowest
message inputs		high			lowest	
message outputs			high	lowest		

Figure 5.5: Best and worst treatments learned by TAR2.

of the attributes. For example, a quick first run of TAR2 indicates that when the number of transition inputs that are states from other machines (see output attribute 4 in the list on page 43) is high, the percentage of OR-nodes reached increases. Figure 5.4 shows a plot of the number of this type of transition inputs verses the percentage of OR-nodes reached. On the left, where the transition input value is low, we see the whole range of percentages reached; but when the transition input value is high we see only the higher range. This agrees with the result from TAR2: if we limit the input to cases in which the number of *state inputs* is high, we avoid the lower left corner, and there is a much better chance we get a high plateau.

The real power of the TAR2 treatment learning approach is in more complex treatments, which suggest restrictions on multiple attributes. Figure 5.5 shows a summary of results

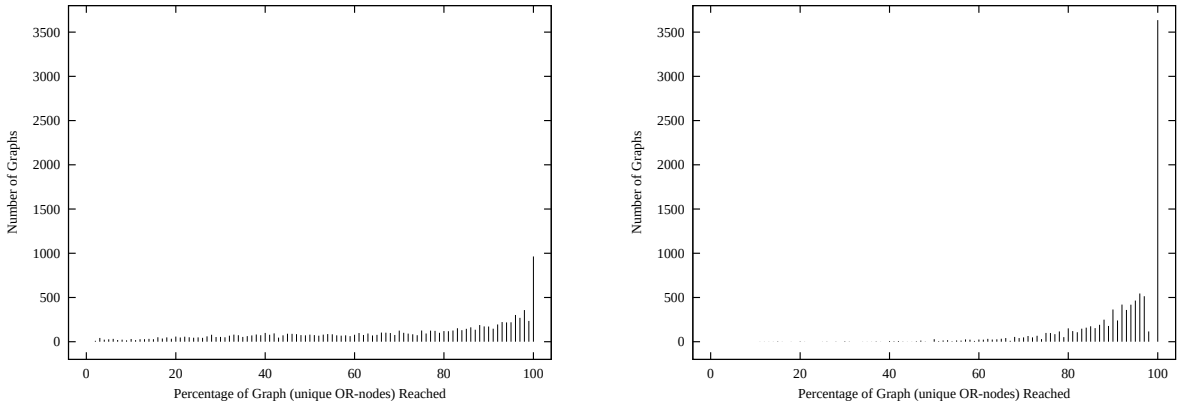


Figure 5.6: Comparison of plateau height for original data (left) and new input models generated according to TAR2's suggested treatments; former average plateau height (as in Figure 5.3) = 53.03%, and the average for new models = 79.37%.

from a series of experiments, in which we tried to determine which combinations of attribute ranges (each treatment considers 4 attributes) are favorable for testability and which give us very untestable graphs. Surprisingly, the three top parameters are low for not only highly testable graphs, but also for graphs that are very difficult to test (the number of finite-state machines and the total number of states are more significant than the total number of transitions). So if we restrict our sample to simpler models (fewer machines, fewer states, fewer transitions) the testability results are polarized.

5.3.1 Testability Results for Models Restricted to High-Testability Attributes

The bottom half of Figure 5.5 shows which attributes have the greatest affect on testability, given that the top three are held low. The most significant attribute is *state inputs,* followed by *message inputs* and *message outputs.* To verify the result from TAR2, we need to make sure that the treatments learned apply generally, not just to the data from the original experiment. Figure 5.6 shows a comparison of plateau height (our indicator of testability) for the original data (left) and a new 10,000 input models (right) generated with the input parameters listed below, which have been modified from the original set to reflect TAR2's suggested treatements. Figure 5.6 shows what we expect: a clear improvement in testability when we follow TAR2's advice.

1. Between 2 and 5 individual finite-state machines.
2. Between 4 and 49 states.
3. Between 0 and 43 transitions.
4. Between 0 and 247 state inputs.
5. Between 0 and 10 messages.
6. Between 0 and 229 message inputs.
7. Between 0 and 241 message outputs.

5.3.2 Interpretation of TAR2 Results

In summary, we now have (1) an automatic method for measuring the testability of communicating finite-state machine input models—a series of random searches yielding time-to-plateau and plateau-height values; we have (2) a good idea of what changes in a particular finite-state model would make that model testable—if we can construct an equivalent model with fewer machines, fewer states, more transition inputs that are states from other machines, etc., it will likely be more testable; and we have (3) some interesting results that could be applied to software design in general.

On point (3) above, first of all, our experiments indicate that smaller models are not necessarily more testable. Larger, more complex models are likely to fall in the middle-to-high testability range, and small, simple models are likely to be either very testable or very difficult to test. It is *connectedness* (the number of transition inputs and outputs), not size, that is most important for testability. Also, we found that transition inputs that are states from other machines are more significant to testability than inputs that are messages passed between machines. A designer may often have a choice of whether information should be available globally (states in one machine visible to another) or should be passed to a specific destination and hidden from others. Usually global variables are thought of as a design liability; the scope of information is to be limited as much as possible in order to limit the propagation of errors. But our experiments show there is actually a tradeoff here: the designer may need to choose whether to make errors easy to find (testability) or less catastrophic when they do occur. The *exception handling* capability of some languanges addresses this principle to some extent, but can only deal with a set of anticipated errors—there will always be strange problems not caught.

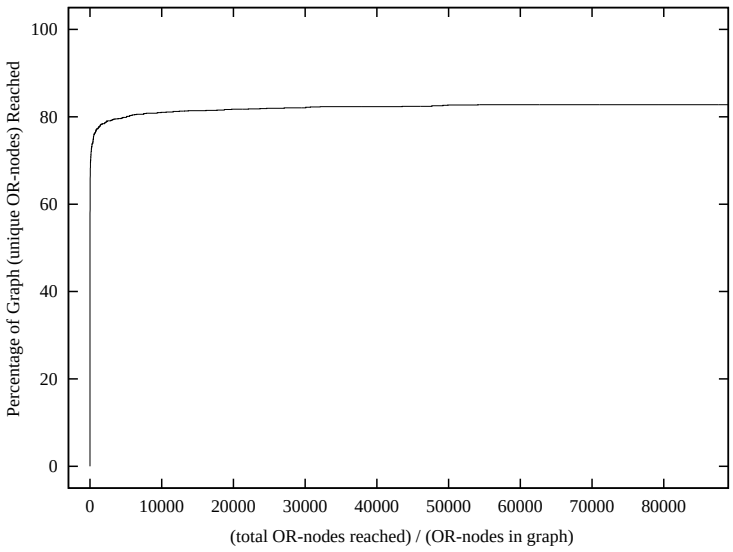


Figure 5.7: Search Results for model with 250 FSM's and 1455 local states (composite size bounded at 2.6516×10^{178} states).

5.4 Search Results for a Very Large Randomly Generated Model

Figure 5.7 shows search results for a very large randomly generated model. The original system of communicating finite-state machines had 250 machines, with a total of 1,455 local states in the individual machines. The size of the composite finite-state machine required to represent this model, 2.6516×10^{178}, is much greater than the state of the art reported in model checking: composite finite-state machine sizes bounded at around 10^{120} states [7]. These are extremely large numbers, and one might wonder whether any real models approach this size. But our 10^{178}-state system started out as just 250 machines with about 6 states each. Many real programs are much bigger than this, with thousands of variables, each with a huge range. So it is definitely worth pursuing alternatives to model checking. Based on experiments like the one shown above in Figure 5.7, we believe the NAYO random search is a promising verification alternative.

Chapter 6

Conclusion

6.1 Summary

To quickly produce high-quality software, we need good tools for measuring software attributes. The testability of software is especially important—we need to know how much confidence to place in test results, how much testing should be done, etc. Also, if we can determine how program structure influences testability, we can design systems so that they will be highly testable.

Abductive random search has performed nearly as well as (and sometimes even better than) time and resource-intensive complete search on the NP-hard problem of searching a knowledge-based system with contradictory information. Menzies et.al. interpret the success of random search in this particular area as a general result that may be applicable to many other kinds of software. Their core idea is that most software systems contain *funnels*: a few key variables largely determine the behavior of the entire program. In these systems, a small number of quick random searches of the execution space will inevitably find the funnel and therefore all (or nearly all) possible program behavior.

During the last ten years model checking techniques have been widely used to verify the correctness of software systems represented by *communicating finite-state machines.* As part of the verification process, these tools must (unfortunately) build a composite finite-state machine representing all possible behavior of the individual machines from the individual model—all of the complex interactions possible between the individual machines. The phrase *state space explosion* characterizes the huge number of states that may be required by the composite finite-state machine. Years of hard work on this problem have produced many new strategies, but in general software verification by model checking is still limited to very small or very carefully designed models.

In this thesis, we have attempted to show how abductive random search might do, for the communicating finite-state machines of model checking, what it has done for knowledge-based systems represented by an AND-OR graph. We begin by showing how communicating finite-state machine models can be automatically translated into equivalent AND-OR graphs. Next, we develop an abductive random search procedure in three stages:

1. A simple search that takes as input a set of nodes, which represent (perhaps contradictory) states from the individual machines in the original model, and outputs a new consistent set of nodes, which is itself consistent with at least part of the input and is as large as possible.

2. A search that progresses in time through a possible execution of the program model (at each step from one point in time to the next we build a consistent set of nodes representing a partial description of a state in the composite finite-state machine model—without ever actually building the composite, which means we avoid the state space explosion problem inherent in model checking).

3. A modified version of the search just described that incorporates features necessary for modeling messages passed between communicating finite-state machines (this type of message passing is used in models of communication protocols).

We believe random search has the potential to do much more than measure testability for these models, and conclude Chapter 4 by using it to verify a simple safety property for a model representing Dekker's solution to the two-process mutual exclusion problem. In these experiments, we reached a plateau quickly just as before. On a correct version of the model, the random search was not able to find the node representing the safety violation. When an error was added to the model, the random search found the safety violation quickly in every case.

We do not attempt a formal analysis of the random search procedure, but instead show experimental results from (1) a model based on an SCR specification, the *Space Shuttle Liquid Hydrogen Subsystem,* and (2) a finite-state model of TCP (transport control protocol). The results of these experiments show that the number of unique nodes (which represent states in the individual machines from the original model) reached rises quickly to a plateau and then remains there indefinitely. The plateau shape is indicative of the *funnel* result we expect: the random search quickly finds nodes representing the key variables and therefore quickly finds all the nodes it is capable of finding. From that point on, it just keeps finding the same information over and over, so the plateau never rises beyond its initial peak.

We interpret the results of these general (input is some random set of nodes) random searches as a kind of testability measure. If the random search is able to reach a higher

plateau more quickly, we infer that the system represented by the AND-OR graph is more testable (the higher plateau indicates that testing will tell us a lot; the fact that a plateau is reached quickly indicates that few tests will be required). With this in mind, we generate a large number of random AND-OR graphs with structure similar to those that would be produced by automatic translation from finite-state machines. Searching these graphs, we find the *time-to-plateau* is always relatively short. So we focus on the height of the plateau—that is, the percentage of OR-nodes reached—as the key indicator of testability.

We then use the TAR2 machine learning tool to go deeper into the search data for these random graphs. It turns out that small and relatively simple models tend to be either highly testable or very difficult to test (our results for more complex models are not so polarized). The most significant factor in determining testability for these models is, in terms of finite-state machine attributes, the total number of transition inputs that are states from other finite-state machines. The number of these *state inputs* is more significant than another attribute, the total number of transition inputs that are messages passed from some other machine. This is an interesting result, because it points out a tradeoff present in software design strategies: information-hiding techniques (i.e., using *message inputs* rather than *state inputs,* which would be analogous to global variables) designed to limit error propagation also diminish testability. Certain programming languages have *exception handling* features that attempt to make errors easy to find without making it easy for them to propagate, but this only works for errors of certain expected types. In general the tradeoff between information hiding and high testability will still be present.

6.2 Future Work

Our AND-OR graph translation scheme, it its present form, requires that state machines representing common discrete variable types be written directly, which is generally not practical (we tend to use only Boolean variables, so that the finite-state machines has only two states). Model checking input languages like SPIN's Promela, allow the designer to work at a higher level, handling not only the variables but typical mathematical operations behind the scenes (to express the operation $i++$ as a set of transitions, even if we restrict i to just 256 possible values, would be extremely tedious). So we suggest future work be done to add these features to the AND-OR graph translation scheme and to confirm with a new set of experiments that the results presented here apply to the wider range of models that could be translated.

Another feature of Promela that is especially important for modelling protocols is the ability to define finite message passing *channels.* Channels are bounded queues representing the salient features of a transmission medium. It is possible to write directly a finite-state machine representing a Promela channel, but, as with variables and mathematical operations, it is very tedious. So we suggest adding something like the channel type to our translation procedure. This would make it possible to translate and search many more models, and so again it would make sense to repeat the search experiments and make sure that our testability results apply to models with channels.

The acme of future work in this area would be to produce a random search-based temporal logic model checker, which would be capable of verifying temporal properties for communicating finite-state machine models (the verification would not be exhaustive, but would have a degree of certainty based on the number of searches performed and the model's testability). The work presented at the end of the Chapter 4 shows how our random search scheme can be used to find a simple property to be true at a certain point in time. The search procedure would need to be expanded to check whether some property is true at time i and then some other property true at time $i + 1$ (in general, whether any set of nodes can be reached at any time, and how the sets reachable at different times are related to each other). It would also be important to build into the translation procedure the ability to go from temporal logic queries to an AND-OR graph, which might involve adding featurers to the AND-OR graph. This new verification tool, since it works on an AND-OR graph, would not need to construct the composite finite-state model and would therefore avoid the state space explosion problem—it would be a real breakthrough.

Appendix A

Code

A.1 AWK Script to Draw Finite-State Machines

This program inputs a set of communicating finite-state machines and outputs them in a form accepted by the program *dot,* which can produce a PostScript picture of the finite-state machines.

```
BEGIN {

    FS = ";"
    node_count = 0
    new_fsm_flag = 0

    /* setup dot file */

    print "digraph fsm"
    print "{"
    print "\tcenter=true;"
    print "\tratio=fill;"
    print "\tsize=\"7.5,10\";"
}

/* if a new machine is encountered, start a definition of its subgraph */

/begin( .*)?/ {

    fsm = substr($0, 7)

    print ""
    print "\tsubgraph cluster" fsm
    print "\t{"
    print "\t\tstyle=dotted;"

    new_fsm_flag = 1
}

/* at the end of a machine's definition, finish its subgraph */
```

```
/end( .*)?/ {

    print "\t}"
}

/* transition definition */

/.+;[ \t]*.+;[ \t]*.+;[ \t]*.+;/ {

    gsub(/[ \t]+/, "", $0)

    if (new_fsm_flag == 1)
    {
        fsm_start = node_count + 1
        new_fsm_flag = 0
    }

    /* see if the parent (current state) node already exists */

    parent_found = 0

    for (i = fsm_start; i <= node_count && parent_found == 0; i++)
    {
        if (nodes[i] == $1)
        {
            parent = i
            parent_found = 1
        }
    }

    /* if parent doesn't exist, make it */

    if (parent_found == 0)
    {
        parent = ++node_count
        print "\t\tnode"   node_count " [label=\"" $1 "\", shape=circle];"
        nodes[node_count] = $1
    }

    /* see if child (next state) exists */

    child_found = 0

    for (i = fsm_start; i <= node_count && child_found == 0; i++)
    {
        if (nodes[i] == $4)
        {
            child = i
            child_found = 1
        }
    }

    /* if child doesn't exist, make it */
```

```
    if (child_found == 0)
    {
        child = ++node_count
        print "\t\tnode" node_count " [label=\"" $4 "\", shape=circle];"
        nodes[node_count] = $4
    }

    /* draw an edge, from parent to child, labeled <input> / <output> */

    print "\t\tnode" parent "->node" child " [label=\"" $2 " / " $3 "\"];"
}

END {

    /* finish off the dot file */

    print "}"
}
```

A.2 Final Version JAVA Translator

This program outputs a NAYO graph representing a set of communicating finite-state machines in the input file. It can also generate an input file for the program *dot* to draw a PostScript picture of the NAYO graph.

```
import java.io.*;
import java.util.*;

public class Tran
{
    static boolean dot = false;         // if true, output .dot file

    static String inputFile;
    static String outputFile;
    static String dotOutputFile;        // file names

    static String fstName;              // remember the name of current
                                        // finite-state machine

    static boolean inAnFst = false;     // within or between machine
                                        // definitions

    static boolean firstTran = false;   // first transition in current machine

    static ArrayList states;
    static ArrayList nodes;             // states in current machine, nodes in
                                        // entire NAYO graph

    static int andCount = 0;            // number of AND-nodes so far
```

```
static String currentState;
static String[] inputs;
static String[] outputs;
static String nextState;            // description of current transition

static PrintWriter output;
static PrintWriter dotOutput;

Tran(String s1, String s2)
{
    inputFile = s1;
    outputFile = s2;
    Tran f = new Tran();
}

Tran()
{
    String s;
    StringTokenizer st1;
    StringTokenizer st2;
    int j;
    nodes = new ArrayList();

    try
        {
            BufferedReader input = new BufferedReader(
                new InputStreamReader(new FileInputStream(inputFile)));

            output = new PrintWriter(new BufferedWriter(
                new FileWriter(outputFile)));

            if (dot)  // setup dot file
                {
                    dotOutput = new PrintWriter(new BufferedWriter(
                        new FileWriter(dotOutputFile)));

                    dotOutput.println("digraph nayo");
                    dotOutput.println("{");
                    dotOutput.println("\tcenter=true;");
                    dotOutput.println("\tratio=fill;");
                    dotOutput.println("\tsize=\"7.5,10\";");
                    dotOutput.println();
                }

            while ((s = input.readLine()) != null)
                {
                    // for a new machine, set flags, get its name, and
                    // setup an array to hold all its state names.

                    if (s.startsWith("begin"))
                        {
                            inAnFst = true;
                            firstTran = true;
                            fstName = s.substring(4);
```

```
        states = new ArrayList();
    }

// the end of the machine's defintion

else if (s.startsWith("end"))
    inAnFst = false;

// ignore anything that's not part of some machine's
// definition

else if (inAnFst)
    {
        st1 = new StringTokenizer(s, " \t;");

        // transition definition

        if (st1.countTokens() == 4)
            {
                for (int i = 0; i < 4; i++)
                    {
                        // 1st column: current state

                        if (i == 0)
                            currentState = st1.nextToken();

                        // 2nd column: inputs

                        else if (i == 1)
                            {
                                st2 = new StringTokenizer(
                                    st1.nextToken(), ",");
                                j = st2.countTokens();
                                inputs = new String[j];

                                // may be many inputs

                                for (int k = 0; k < j; k++)
                                    inputs[k] = st2.nextToken();
                            }

                        // 3rd column: outputs

                        else if (i == 2)
                            {
                                st2 = new StringTokenizer(
                                    st1.nextToken(), ",");
                                j = st2.countTokens();
                                outputs = new String[j];

                                // may be many outputs

                                for (int k = 0; k < j; k++)
                                    outputs[k] = st2.nextToken();
```

```
                                        }

                                    // 4th column: next state

                                    else if (i == 3)
                                        nextState = st1.nextToken();
                                }

                            // once we have all the information,
                            // add it to the NAYO graph

                            transition();
                        }

                    // if only one column, it's a state declaration

                    else if (st1.countTokens() == 1)
                        {
                            currentState = st1.nextToken();

                            // make the appropriate NO-edges

                            stateDeclaration();
                        }
                }
            }

        // finish dot file

        if (dot)
            {
                dotOutput.println("}");
                dotOutput.close();
            }

        input.close();
        output.close();
    }
catch (IOException e)
    {
    }
}

void transition()
{
    int inIndex;
    int outIndex;

    // if this is the first transition in the machine, mark it as a
    // default input

    if (firstTran == true)
        {
            output.println("i " + nodes.indexOf(currentState) + " " + currentState);
```

```
         firstTran = false;
    }

// make an AND-node for the transition

int andIndex = nodes.size();
nodes.add("and" + ++andCount);

// make current state a parent of the AND-node

if (dot)
    {
        dotOutput.println("\t\"" + andIndex  + "\" [style=\"filled\"];");
        dotOutput.println("\t\"" + nodes.indexOf(currentState) + "\"->\""
                          + andIndex + "\";");
    }

output.println("a " + andIndex + " and" + andCount);
output.println("y " + nodes.indexOf(currentState) + " " + andIndex);

// if there are inputs listed, make them parents of the AND-node

if (!(inputs[0].equals("-")))

    for (int i = 0; i < inputs.length; i++)
        {
            if (!(nodes.contains(inputs[i])))
                {
                    inIndex = nodes.size();
                    nodes.add(inputs[i]);

                    output.println("o " + inIndex + " " + inputs[i]);
                }
            else
                inIndex = nodes.indexOf(inputs[i]);

            if (dot)
                dotOutput.println("\t\"" + inIndex + "\"->\"" + andIndex + "\";");

            output.println("y " + inIndex + " " + andIndex);
        }

// if there are outputs listed, make them children of the AND-node

if (!(outputs[0].equals("-")))
    {
        for (int i = 0; i < outputs.length; i++)
            {
                if (!(nodes.contains(outputs[i])))
                    {
                        outIndex = nodes.size();
                        nodes.add(outputs[i]);

                        output.println("o " + outIndex + " " + outputs[i]);
```

```
                        }
                    else
                        outIndex = nodes.indexOf(outputs[i]);

                    if (dot)
                        dotOutput.println("\t\"" + andIndex + "\"->\""
                                          + outIndex + "\";");

                    output.println("y " + andIndex + " " + outIndex);
                }
        }

    // make next state a child of the AND-node

    if (dot)
        dotOutput.println("\t\"" + andIndex + "\"->\""
                          + nodes.indexOf(nextState) + "\";");

    output.println("y " + andIndex + " "
                   + nodes.indexOf(nextState));
}

void stateDeclaration()
{
    int csIndex;
    int j = states.size();

    // if this state is not yet in the list, add it and make an OR-node
    // for it

    if ((csIndex = nodes.indexOf(currentState)) == -1)
        {
            csIndex = nodes.size();
            nodes.add(currentState);

            output.println("o " + csIndex + " " + currentState);
        }

    // make NO-edges from this state to all others in its machine

    if (dot)
        for (int i = 0; i < j; i++)
            dotOutput.println("\t\"" + csIndex
                              + "\"->\"" + ((Integer)states.get(i)).intValue()
                              + "\" [label=\"no\",style=\"dotted\","
                              + "dir=\"both\"];");

    for (int i = 0; i < j; i++)
        output.println("n " + csIndex + " "
                       + ((Integer)states.get(i)).intValue());

    // add this state to the list of states for this machine

    states.add(new Integer(csIndex));
```

```
    }

    public static void main(String args[])
    {
        // input is: java Tran [-dot] <input> <output> [dot file]

        if (args[0].equals("-dot"))
            {
                dot = true;
                inputFile = args[1];
                outputFile = args[2];
                dotOutputFile = outputFile + ".dot";
                Tran f = new Tran();
            }
        else
            {
                inputFile = args[0];
                outputFile = args[1];
                Tran f = new Tran();
            }
    }
}
```

A.3 Random Search Program

This program performs a series of random searches on a NAYO graph. Output may be a summary (unique nodes reached vs. total nodes processed) or more specific (which nodes were reached, and at what time were they first reached).

```
import java.util.*;

public class Node3
{
    ArrayList yesList;
    ArrayList noList;
    int type;          // static info about the node


    int waitValue;
    int initWait;
    int disqualified;
    int tempFound;
    int found;         // fields manipulated by the search---waitValue
                       // decremented when a parent is processed, = 0 when
                       // node is reached; disqualified = true when node
                       // should not be processed; tempFound: found at
                       // current time; found: found at any time (and is
                       // set to the earliest time the node is found)

    Node3(int t)
    {
```

```
        type = t;

        if (type == 2)
            initWait = -1;
        else initWait = 1;

        noList = new ArrayList();
        yesList = new ArrayList();
        waitValue = -1;
        disqualified = -1;
        tempFound = -1;
        found = -1;
    }
}

import java.io.*;
import java.util.*;

public class Walk4
{
    static String inputFile;
    static String statFile;
    static String statString;
    static String outputFile;              // file names; "stat" is statistics about
                                           // the NAYO graph if it comes from the
                                           // random NAYO generator program

    static ArrayList nayo;                 // list of nodes
    static ArrayList queue;                // list of node indices (integers)

    static int walkTime;                   // manipulated by search
    static int maxTime;                    // max. value of walkTime

    static RandomNumberGenerator r;
    static BufferedReader input;
    static PrintWriter output;

    static int numSearches;
    static int searchTime;
    static int stat;
    static int statOr;
    static int saveHeight = 0;
    static int saveTime = 0;               // misc. record keeping info

    static boolean randomNayos = false;    // was NAYO randomly generated?
    static boolean verbose = false;

    Walk4(String in, int mt, int ns, String sf, String of)
    {
        randomNayos = true;
        inputFile = in;
        maxTime = mt;
        numSearches = ns;
```

```
    // get info about NAYO graph from ‘‘stat’’ file

    try
        {
            input = new BufferedReader(new InputStreamReader(
                new FileInputStream(sf)));

            String temp;

            while ((temp = input.readLine()) != null)
                statString = temp;
        }
    catch(IOException e)
        {
        }

    outputFile = of;

    stat = 0;
    statOr = 0;
    nayo = new ArrayList();
    queue = new ArrayList();
    r = new RandomNumberGenerator();

    readFile();
    loopSearch();
}

Walk4()
{
    int i;
    Node3 n1;
    Node3 n2;
    Node3 n3;

    boolean stop = false;
    walkTime = 0;

    // outer search loop

    while ((stop == false) /* && (walkTime <= maxTime) */)
        {
            stop = true;

            // if there are nodes in the queue, remove the head and
            // process it

            while (!(queue.isEmpty()))
                {
                    i = ((Integer)queue.remove(0)).intValue();
                    n1 = (Node3)nayo.get(i);

                    // if node from queue is already disqualified for this
                    // time tick, throw it out
```

```
if (n1.disqualified != walkTime)
    {
        if (verbose)
            System.out.println(walkTime + "       " + i);

        // reset (consume) values indicating that node
        // from queue was reached

        n1.waitValue = n1.initWait;
        n1.tempFound = walkTime;

        // if it's an OR-node, disqualify it so that
        // it can't be processed again during this time tick

        if (n1.type <= 1)
            n1.disqualified = walkTime;

        // disqualify NO-edge children

        for (int j = 0; j < n1.noList.size(); j++)
            {
                i = ((Integer)n1.noList.get(j)).intValue();
                n2 = (Node3)nayo.get(i);
                n2.disqualified = walkTime;
            }

        // find YES-edge children

        for (int j = 0; j < n1.yesList.size(); j++)
            {
                i = ((Integer)n1.yesList.get(j)).intValue();
                n2 = (Node3)nayo.get(i);

                // if a YES-edge child not yet
                // reached, decrement child's waitValue

                if (n2.waitValue > 0)
                    {
                        n2.waitValue--;

                        if (verbose)
                            System.out.println(walkTime
                                + "           (" + i + ") "
                                + n2.waitValue);

                        // if waitValue goes to 0, the
                        // child is reached; keep track
                        // of some misc. info about
                        // child, mark it found for this
                        // time tick, and put it into the
                        // queue at some random index

                        if (n2.waitValue == 0)
```

```
                                                {
                                                    if (n2.type <= 1) stat++;

                                                    if (verbose)
                                                        System.out.println(walkTime
                                                            + "            " + i);

                                                    if (n2.tempFound != walkTime)
                                                        {
                                                            if (n2.type == 2)
                                                                {
                                                                    n3 = (Node3)nayo.get(
                                                                        n2.stateParent);
                                                                    n3.tempFound = -1;
                                                                }

                                                            n2.tempFound = walkTime;
                                                            enqueue(i);

                                                            if (n2.found != searchTime)
                                                                stop = false;

                                                            if (n2.found == -1)
                                                                n2.found = searchTime;
                                                        }
                                                }
                                            }
                                        }
                                    }
                                }

                    // set up search for next time tick: put this-time reached set
                    // and ``frontier'' of nodes implied by this-time set but
                    // contradicting it in also.

                    if ((stop == false) /* && (walkTime < maxTime)*/)
                        for (int j = 0; j < nayo.size(); j++)
                            {
                                n1 = (Node3)nayo.get(j);

                                if ((n1.tempFound == walkTime) && (n1.disqualified == walkTime))
                                    enqueue(j);

                                else n1.waitValue = n1.initWait;
                            }

                    walkTime++;
                }
        }

        static void loopSearch()
        {
            Node3 n;
            int i = 0;
```

```
int peakTime = 0;
int peakStat = 0;

try
    {
        output = new PrintWriter(new BufferedWriter(
            new FileWriter(outputFile)));

        output.println("0 0");

        // repeat the random search a bunch of times

        for (searchTime = 1; searchTime <= numSearches; searchTime++)
            {
                Walk4 nw = new Walk4();

                if (verbose)
                    System.out.println();

                // setup NAYO graph (list of nodes) for the next
                // search iteration; output current results as a
                // progress indicator

                for (int j = 0; j < nayo.size(); j++)
                    {
                        n = (Node3)nayo.get(j);
                        n.tempFound = -1;
                        n.disqualified = -1;

                        if (n.type <= 1)
                            {
                                if (n.type == 0 || (randomNayos && n.type == 1))
                                    {
                                        n.waitValue = 0;
                                        enqueue(j);
                                    }

                                if (n.found == searchTime)
                                    {
                                        if (n.type == -1)
                                            System.out.println("found " + j + " - "
                                                               + stat + " " + i);

                                        peakTime = searchTime;
                                        peakStat = stat;
                                        i++;
                                    }
                            }
                    }

                output.println(stat + " " + i);
            }

        // output summary results
```

```
            if ((peakTime < (double) searchTime / 2) && (i > 0))
                {
                    // output.println(statString + "\t" + statOr + "\t"
                    //   + (100 * i / statOr) + "\t" + (peakStat / statOr));
                    System.out.println(statString + "\t" + statOr + "\t"
                        + (100 * i / statOr) + "\t" + (peakStat / statOr));
                }

            // (saveHeight and saveTime are static) here we update their values

            if (i > saveHeight)
                saveHeight = i;

            if (peakStat > saveTime)
                saveTime = peakStat;

            output.close();
        }
    catch(IOException e)
        {
        }
}

// put a node's index into the queue at some random (queue) index; don't
// put the same node in twice

static void enqueue(int i)
{
    if (!(queue.contains(new Integer(i))))
        {
            if (verbose) System.out.println("[" + i + "]");
            queue.add((r.nextInt(queue.size() + 1)),
                      new Integer(i));
        }
}

// read the input (NAYO) file and assign nodes the right waitValues

static void readFile()
{
    String s;
    StringTokenizer st;
    String token1;
    String token2;
    String token3;
    Node3 n;
    Integer temp;
    int i;

    try
        {
            input = new BufferedReader(new InputStreamReader(
                new FileInputStream(inputFile)));
```

```
while ((s = input.readLine()) != null)
    {
        st = new StringTokenizer(s, " ");
        token1 = st.nextToken();
        token2 = st.nextToken();
        token3 = st.nextToken();

        // an OR-node

        if (token1.equals("o"))
            {
                n = new Node3(1);
                nayo.add(n);

                // for randomly generated NAYOs make all
                // OR-nodes 1st-time input; conflicts will be
                // resolved by search in first time tick

                if (randomNayos)
                    {
                        n.waitValue = 0;
                        enqueue(nayo.size() - 1);
                    }

                // otherwise, make waitValue 1; this means
                // OR-node must be reached via only 1 parent

                else n.waitValue = 1;

                statOr++;  // total number of OR-nodes
            }

        // OR-node specifically designated as an input

        else if (token1.equals("i") && !randomNayos)
            {
                n = (Node3)nayo.get(Integer.parseInt(token2));
                n.type = 0;

                n.waitValue = 0;
                n.found = 1;
                enqueue(Integer.parseInt(token2));
            }

        // OR-node specifically designated to represent a property

        else if (token1.equals("p") && !randomNayos)
            {
                n = new Node3(-1);
                nayo.add(n);
                n.waitValue = 1;
                statOr++;
            }
```

```
// AND-node

else if (token1.equals("a"))
    {
        n = new Node3(2);
        nayo.add(n);

        // do nothing about AND-node's waitValue here,
        // it will be set according to the number of edges
        // going into it
    }

// YES-edge

else if (token1.equals("y"))
    {
        // find the node's involved; add child to
        // parent's list of children via YES-edges

        temp = Integer.decode(token3);
        n = (Node3)nayo.get(Integer.parseInt(token2));
        n.yesList.add(temp);
        n = (Node3)nayo.get(temp.intValue());

        // if child is an AND-node, increment its
        // waitValue (and initWait, which just remembers
        // the node's waitValue from the point at which
        // the search starts)

        if (n.type == 2)
            {
                if (n.waitValue == -1)
                    {
                        // n.stateParent = Integer.parseInt(token2);
                        n.waitValue = n.initWait = 1;
                    }
                else
                    {
                        n.waitValue++;
                        n.initWait = n.waitValue;
                    }
            }
    }

// NO-edge

else if (token1.equals("n"))
    {
        // find the nodes involved, add the child to
        // the parent's list of children via NO-edges

        n = (Node3)nayo.get(Integer.parseInt(token2));
        n.noList.add(Integer.decode(token3));
```

```
                                n = (Node3)nayo.get(Integer.parseInt(token3));
                                n.noList.add(Integer.decode(token2));
                            }
                        }

                    input.close();
                }
            catch (IOException e)
                {
                }
        }

        public static void main(String args[])
        {
            // input is: java Walk4 <input> <max search time ticks> <number of
            // searches> <output>

            int i = 0;

            inputFile = args[i++];
            maxTime = Integer.parseInt(args[i++]);
            numSearches = Integer.parseInt(args[i++]);
            outputFile = args[i];

            stat = 0;
            statOr = 0;
            nayo = new ArrayList();
            queue = new ArrayList();
            r = new RandomNumberGenerator();

            readFile();
            loopSearch();
        }
}
```

A.4 Random NAYO Generator

This program randomly generates NAYO graphs according to parameters describing a set of communicating finite-state machines equivalent to the NAYO graph. The NAYO can then be searched by the random search program listed above. This program can also generate a file for the program *dot,* which creates a PostScript picture of the NAYO graph.

```
import java.io.*;
import java.util.*;
import java.math.*;

public class RandomNayo2
{
    static RandomNumberGenerator r;
```

```
static int numFsts;                 // number of machines in system

static int maxStates;               // max states per machine

static int maxTrans;                // max transitions per machine

static int maxPars;                 // max inputs (that are states from
                                    // other machines) per transition

static int maxMessages;             // max unique (consumable) messages
                                    // passed between machines
static int maxMessagePars;
static int maxMessageKids;          // max inputs and outputs that are
                                    // (consumable) messages per transition

static boolean dot;
static String outputFile;
static String dotOutputFile;
static String statOutputFile;
static PrintWriter output;
static PrintWriter dotOutput;
static PrintWriter statOutput;      // miscellaneous I/O stuff, also flag
                                    // to indicate whether dot file should be created

static int scale;                   // output includes approximate bound
                                    // on number of states in the
                                    // equivalent composite finite-state machine.
                                    // This will likely be a huge number,
                                    // so scale is the power of 10 we
                                    // think it might be...

public RandomNayo2(int n, int m, int t, int p, int mm, int mp, int mk,
                   int sc, String s, String f)
{
    numFsts = n;
    maxStates = m;
    maxTrans = t;
    maxPars = p;
    maxMessages = mm;
    maxMessagePars = mp;
    maxMessageKids = mk;
    dot = false;
    scale = sc;
    outputFile = s;
    statOutputFile = f;

    RandomNayo2 rn = new RandomNayo2();
}

RandomNayo2()
{
    r = new RandomNumberGenerator();

    int par;
```

```
int kid;                                            // input and output
                                                    // (new for each transition)

int andCount = 0;
int nodeCount = 0;
int messages;                                       // number of
                                                    // AND-nodes, total
                                                    // nodes, message nodes

int[] stateCount = new int[numFsts];
int totalStates = 0;
int[] firstState = new int[numFsts];
int[] transCount = new int[numFsts];
int totalTrans = 0;
int[][] parCount = new int[numFsts][];
int totalPars = 0;
int[][] messageParCount = new int [numFsts][];
int totalMessagePars = 0;
int[][] messageKidCount = new int [numFsts][];
int totalMessageKids = 0;
int[] randFsts;
int[] rand;
int randIndex;                                      // set up a bunch of
                                                    // per-machine and per-transition info
                                                    // arrays, also ints to keep track of totals

boolean[] skip = new boolean[numFsts];              // flag marks machine
                                                    // in which a transition
                                                    // is defined as off-limits for
                                                    // inputs and outputs

// keep track of info for equivalent composite finite-state machine

BigDecimal compStates = new BigDecimal(new BigInteger("1"), scale);
BigDecimal transMultiplier = new BigDecimal(new BigInteger("1"), scale);
BigDecimal compTrans = new BigDecimal(new BigInteger("0"), scale);

try
    {
        output = new PrintWriter(new BufferedWriter(
            new FileWriter(outputFile)));

        if (dot)
            {
                dotOutput = new PrintWriter(new BufferedWriter(
                    new FileWriter(dotOutputFile)));

                // set up dot file

                dotOutput.println("digraph RandomNayo2");
                dotOutput.println("{");
                dotOutput.println("\tcenter=true;");
                dotOutput.println("\tratio=fill;");
                dotOutput.println("\tsize=\"7.5,10\";");
```

```
        dotOutput.println();
    }

statOutput = new PrintWriter(new BufferedWriter(
    new FileWriter(statOutputFile)));

// set the number of unique (consumable) messages and make a
// node for each

messages = r.nextInt(maxMessages + 1);

for (int i = 0; i < messages; i++)
    {
        if (dot)
            dotOutput.println("\tor" + nodeCount + ";");

        output.println("o " + nodeCount + " or" + nodeCount++);
    }

// for each machine...

for (int i = 0; i < numFsts; i++)
    {
        // it must have a least one (the first) state

        firstState[i] = nodeCount;

        // set the total number of states

        stateCount[i] = 2 + r.nextInt(maxStates - 1); // at
                                                      // least 2

        // update the number of states required by the
        // composite (multiply previous value by the number in
        // this machine)

        compStates = compStates.multiply(new BigDecimal(
            (new Integer(stateCount[i])).toString()));

        // update the total number of LOCAL states (from the
        // individual machines)

        totalStates += stateCount[i];

        // make an OR-node for the 1st state in the machine,
        // mark it with ``i'' as a (default) input

        if (dot)
            dotOutput.println("\tor" + nodeCount + ";");

        output.println("o " + nodeCount + " or" + nodeCount);
        output.println("i " + nodeCount + " or" + nodeCount);

        // make OR-nodes for the rest of the states in this
```

```
            // machine

            for (int j = 1; j < stateCount[i]; j++)
                {
                    output.println("o " + (nodeCount + j) + " or"
                                   + (nodeCount + j));

                    // link all states within this machine by
                    // NO-edges

                    for (int k = 0; k < j; k++)
                        {
                            if (dot)
                                dotOutput.println("\tor" + (nodeCount + j) + "->or"
                                    + (nodeCount + k)
                                    + " [label=\"no\",style=\"dotted\","
                                    + "dir=\"both\"];");

                            output.println("n " + (nodeCount + j) + " "
                                           + (nodeCount + k));
                        }
                }

            if (dot)
                dotOutput.println();

            // update record keeping info...

            nodeCount += stateCount[i];
            transCount[i] = r.nextInt(maxTrans + 1);
            totalTrans += transCount[i];
            parCount[i] = new int[transCount[i]];
            messageParCount[i] = new int[transCount[i]];
            messageKidCount[i] = new int[transCount[i]];
        }

    // for each machine...

    for (int i = 0; i < numFsts; i++)
        {
            // make some transitions

            for (int j = 0; j < transCount[i]; j++)
                {
                    // in order to update the number of
                    // transitions that would be required by the
                    // composite, set up a BigDecimal to hold the value

                    transMultiplier = new BigDecimal(new BigInteger("1"), scale);

                    // get a shuffled list of the states in this machine

                    rand = shuffle(firstState[i], stateCount[i]);
```

```
// don't skip any machines...

for (int k = 0; k < numFsts; k++)
    skip[k] = false;

// except this one (when looking for
// transition inputs that are states)

skip[i] = true;

// get a current and a next state from the
// shuffled list of states within this machine

par = rand[0];
kid = rand[1];

// record keeping...

if (maxPars > 0)
    parCount[i][j] = 1 + r.nextInt(maxPars);
else parCount[i][j] = 0;

totalPars += parCount[i][j];
messageParCount[i][j] = r.nextInt(maxMessagePars + 1);
totalMessagePars += messageParCount[i][j];
messageKidCount[i][j] = r.nextInt(maxMessageKids + 1);
totalMessageKids += messageKidCount[i][j];

if (parCount[i][j] > numFsts)
    parCount[i][j] = numFsts;

if (messageParCount[i][j] > messages)
    messageParCount[i][j] = messages;

if (messageKidCount[i][j] > messages)
    messageKidCount[i][j] = messages;

// make the transition:

// if just one input, no AND-node is needed,
// just make the input parent of the output

if ((parCount[i][j] == 1) && (messageParCount[i][j] == 0)
    && (messageKidCount[i][j] == 0))
    {
        if (dot)
            dotOutput.println("\tor" + par + "->or"
                + kid + ";");

        output.println("y " + par + " " + kid);
    }

// otherwise make the AND-node...
```

```
else
    {
        randIndex = 0;
        randFsts = shuffle(0, numFsts);

        // make the current state a parent of
        // the AND-node

        if (dot)
            {
                dotOutput.println("\tor" + par + "->and"
                                  + andCount + ";");
                dotOutput.println("\tand" + andCount + "->or"
                                  + kid + ";");
            }

        output.println("a " + nodeCount + " and" + andCount);
        output.println("y " + par + " " + nodeCount);
        output.println("y " + nodeCount + " " + kid);

        // get (random) state inputs and make
        // them parents of the AND-node

        for (int k = 1; k < parCount[i][j]; k++)
            {
                if (randFsts[randIndex] == i)
                    randIndex++;

                skip[randFsts[randIndex]] = true;
                rand = shuffle(firstState[randFsts[randIndex]],
                    stateCount[randFsts[randIndex++]]);

                par = rand[0];

                if (dot)
                    {
                        dotOutput.println("\tor" + par
                            + "->and" + andCount + ";");
                        dotOutput.println("\tand" + andCount
                            + "->or" + par + ";");
                    }

                output.println("y " + par + " " + nodeCount);
                output.println("y " + nodeCount + " " + par);
            }

        // get a shuffled list of the set of
        // possible (consumable) messages

        randIndex = 0;
        rand = shuffle(0, messages);

        // get input messages; make them
        // parents of the AND-node
```

```
                            for (int k = 0; k < messageParCount[i][j]; k++)
                                {
                                    par = rand[randIndex++];

                                    if (dot)
                                        dotOutput.println("\tor" + par + "->and"
                                            + andCount + ";");

                                    output.println("y " + par + " " + nodeCount);
                                }

                            // get a new shuffled list of the messages

                            randIndex = 0;
                            rand = shuffle(0, messages);

                            // get output messages; make them kids
                            // of the AND-node

                            for (int k = 0; k < messageKidCount[i][j]; k++)
                                {
                                    kid = rand[randIndex++];

                                    if (dot)
                                        dotOutput.println("\tand" + andCount
                                                          + "->or" + kid + ";");

                                    output.println("y " + nodeCount + " " + kid);
                                }

                            nodeCount++;
                            andCount++;  // record keeping...
                        }

                    // update numbers for composite size

                    for (int k = 0; k < numFsts; k++)
                        if (!skip[k])
                            transMultiplier = transMultiplier.multiply(
                                new BigDecimal(
                                (new Integer(stateCount[k])).toString()));

                    compTrans = compTrans.add(transMultiplier);
                }
            }

        // finish dot file

        if (dot)
            {
                dotOutput.println("}");
                dotOutput.close();
            }
```

```
            output.close();

            // put NAYO graph info into ‘‘stat" file

            statOutput.println(numFsts + "\t" + totalStates + "\t"
                + totalTrans + "\t" + messages + "\t" + totalPars + "\t"
                + totalMessagePars + "\t" + totalMessageKids);

            statOutput.close();

        }
    catch(IOException e)
        {
        }
}

// return an array of shuffled ints between base and max

int[] shuffle(int base, int max)
{
    int j;
    int[] rand = new int[max];

    for (int i = 0; i < max; i++)
        rand[i] = -1;

    for (int i = 0; i < max; i++)
        {
            j = r.nextInt(max);

            while(rand[j] != -1)
                {
                    if (j < max)
                        j++;

                    if (j == max)
                        j = 0;
                }

            rand[j] = i + base;
        }

    return rand;
}

public static void main(String args[])
{
    // input is: java RandomNayo2 [-dot] <number of machines> <max states
    // per machine> <max transitions per machine> <max inputs per tranisition
    // that are states from some other machine> <max unique (consumable)
    // messages> <max inputs per transition that are messages> <max
    // outputs per transition that are messages> <guess at composite size (power
    // of 10)> <output for NAYO graph> <output for NAYO information>
```

```
        // note: name of dot file is just <output>.dot

        int i = 0;

        if (args[0].equals("-dot"))
            {
                dot = true;
                dotOutputFile = args[8] + ".dot";
                i++;
            }
        else
            dot = false;

        numFsts = Integer.parseInt(args[i++]);
        maxStates = Integer.parseInt(args[i++]);
        maxTrans = Integer.parseInt(args[i++]);
        maxPars = Integer.parseInt(args[i++]);
        maxMessages = Integer.parseInt(args[i++]);
        maxMessagePars = Integer.parseInt(args[i++]);
        maxMessageKids = Integer.parseInt(args[i++]);
        scale = Integer.parseInt(args[i++]);
        outputFile = args[i++];
        statOutputFile = args[i];

        RandomNayo2 rn = new RandomNayo2();
    }
}
```

Appendix B

Example Systems

B.1 SCR: Space Shuttle Liquid Hydrogen Subsystem

B.1.1 SCR Specification from [2]

Type Dictonary

Name	Base Type	Units	Legal Values
yPressure	Float	Psia	$[0.0, 43.7]$
yTime	Integer	second	$[0, 405]$
yValve	Enumerated	N/A	open, closed
yVolume	Integer	%	$[0, 100]$

Mode Class Dictionary

Name	Modes	Initial Mode	Table?	Comment
smSystem	sChilldown, sSlowFill, sFastFill, sTopping sReplenish	sChilldown	Yes	models the LH2 system

Monitored Variable Dictionary

Name	Type	Initial Value	Accuracy	Comment
mPressureState	yPressure	0.0	N/A	Tried modeling this as another mode class, but had trouble with crossing modes. Does this start at 0?
mTimeState	yTime	0	N/A	Tried previously as a mode class
mVolumeState	yVolume	0	N/A	Tride previously as a mode class

Controlled Variable Dictionary

Name	Type	Initial Value	Accuracy
cAuxiliaryFill	yValve	closed	N/A
cChilldown	yValve	closed	N/A
cExternalTankVent	yValve	closed	N/A
cFillDisconnect	yValve	open	N/A
cHighPointBleed	yValve	closed	N/A
cInboardFillDrain	yValve	open	N/A
cMainFill	yValve	closed	N/A
cMainFillRedu	yValve	closed	N/A
cOutboardFillDrain	yValve	open	N/A
cPreValve	yValve	open	N/A
cRecirculationDisconnect	yValve	closed	N/A
cRecirculationPreValve	yValve	closed	N/A
cReplenish	yValve	closed	N/A
cTopping	yValve	closed	N/A
cTransfer	yValve	closed	N/A
cTransferLineVent	yValve	open	N/A

Mode Transition Table for smSystem

Source Mode	Events	Destination Mode
sChilldown	@T(mPressureState = 43.7)	sSlowFill
sSlowFill	@T(mVolumeState = 2)	sFastFill
sFastFill	@T(mVolumeState = 98)	sTopping
sTopping	@T(mVolumeState = 100)	sReplenish

Event Table for cTransferLineVent

Source Mode	Events
sChilldown	@T(mTimeState = 60)
	closed

Event Table for cTransferLine

Modes	Events	
sChilldown	@T(mTimeState = 165)	@T(mTimeState = 405)
sSlowFill	@T(mVolumeState = 2)	NEVER
sFastFill	NEVER	@T(mVolumeState = 98)
	open	closed

Event Table for cTopping

Modes	Events
sSlowFill	@T(true) WHEN (cChilldown = open)
	open

Event Table for cReplenish

Modes	Events
sFastFill	@T(true) WHEN (cPreValve = closed)
	open

Event Table for cRecirculationPreValve

Modes	Events
sFastFill	@T(INMODE)
	open

Event Table for cPreValve

Modes	Events
sFastFill	@T(true) WHEN (cRecirculationPreValve = open)
	closed

Event Table for cOutboardFillDrain

Modes	Events
sChilldown	@T(mTimeState = 60)
	open

Event Table for cMainFillRedu

Modes	Events	
sFastFill	@T(true) WHEN (cPreValve = closed)	NEVER
sTopping	NEVER	@T(mVolumeState = 100)
	open	closed

Event Table for cMainFill

Modes	Events	
sChilldown	@T(mTimeState = 165)	NEVER
sTopping	NEVER	@T(mVolumeState = 100)
	open	closed

Event Table for cInboardFillDrain

Modes	Events
sFastFill	@T(true) WHEN (cPreValve = closed)
	closed

Event Table for cHighPointBleed

Modes	Events
sSlowFill	@T(true) WHEN (cChilldown = open AND cTopping = open)
	open

Event Table for cExternalTankVent

Modes	Events	
sChilldown	@T(mTimeState = 60)	@T(mPressureState = 43.7)
sFastFill	@T(mVolumeState = 98)	NEVER
	open	closed

Event Table for cChilldown

Modes	Events	
sChilldown	@T(mTimeState = 165)	@T(mPressureState = 43.7)
sSlowFill	@T(true) WHEN (cChilldown = closed)	NEVER
	open	closed

B.1.2 Specification Written as Finite-State Machines for Translation to NAYO Graph

```
begin smSystem
smSystem=sChilldown;
smSystem=sSlowFill;
smSystem=sFastFill;
smSystem=sTopping;
smSystem=sReplenish;
end smSystem

begin mPressureState
mPressureState=0;
mPressureState=43.7;
end mPressureState

begin mTimeState
mTimeState=0;
mTimeState=60;
mTimeState=165;
```

```
mTimeState=405;
end mTimeState

begin mVolumeState
mVolumeState=0;
mVolumeState=2;
mVolumeState=98;
mVolumeState=100;
end mVolumeState

begin cAuxiliaryFill
cAuxiliaryFill=closed;
cAuxiliaryFill=open;
end cAuxiliaryFill

begin cChilldown
cChilldown=closed;
cChilldown=open;
end cChilldown

begin cExternalTankVent
cExternalTankVent=closed;
cExternalTankVent=open;
end cExternalTankVent

begin cFillDisconnect
cFillDisconnect=open;
cFillDisconnect=closed;
end cFillDisconnect

begin cHighPointBleed
cHighPointBleed=closed;
cHighPointBleed=open;
end cHighPointBleed

begin cInboardFillDrain
cInboardFillDrain=open;
cInboardFillDrain=closed;
end cInboardFillDrain

begin cMainFill
cMainFill=closed;
cMainFill=open;
end cMainFill

begin cMainFillRedu
cMainFillRedu=closed;
cMainFillRedu=open;
end cMainFillRedu

begin cOutboardFillDrain
cOutboardFillDrain=closed;
cOutboardFillDrain=open;
end COutboardFillDrain
```

```
begin cPreValve
cPreValve=open;
cPreValve=closed;
end cPreValve

begin cRecirculationDisconnect
cRecirculationDisconnect=open;
cRecirculationDisconnect=closed;
end cRecirculationDisconnect

begin cRecirculationPreValve
cRecirculationPreValve=closed;
cRecirculationPreValve=open;
end cRecirculationPreValve

begin cReplenish
cReplenish=closed;
cReplenish=open;
end cReplenish

begin cTopping
cTopping=closed;
cTopping=open;
end cTopping

begin cTransferLine
cTransferLine=closed;
cTransferLine=open;
end cTransferLine

begin cTransferLineVent
cTransferLineVent=open;
cTransferLineVent=closed;
end cTransferLineVent

begin smSystem_Mode_Transition_Table
smSystem=sChilldown; mPressureState=43.7; -; smSystem=sSlowFill;
smSystem=sSlowFill; mVolumeState=2; -; smSystem=sFastFill;
smSystem=sFastFill; mVolumeState=98; -; smSystem=sTopping;
smSystem=sTopping; mVolumeState=100; -; smSystem=sReplenish;
end smSystem_Mode_Transition_Table

begin cTransferLineVent_Event_Table
cTransferLineVent=open; smSystem=sChilldown,mTimeState=60; -; cTransferLineVent=closed;
end cTransferLineVent_Event_Table

begin cTransferLine_Event_Table
cTransferLine=closed; smSystem=sChilldown,mTimeState=165; -; cTransferLine=open;
cTransferLine=closed; smSystem=sSlowFill,mVolumeState=2; -; cTransferLine=open;
cTransferLine=open; smSystem=sChilldown,mTimeState=405; -; cTransferLine=closed;
cTransferLine=open; smSystem=sFastFill,mVolumeState=98; -; cTransferLine=closed;
end cTransferLine_Event_Table
```

```
begin cTopping_Event_Table
cTopping=closed; smSystem=sSlowFill,cChilldown=open; -; cTopping=open;
end cTopping_Event_Table

begin cReplenish_Event_Table
cReplenish=closed; smSystem=sFastFill,cPreValve=closed; -; cReplenish=open;
end cReplenish_Event_Table

begin cRecirculationPreValve_Event_Table
cRecirculationPreValve=closed; smSystem=sFastFill; -; cRecirculationPreValve=open;
end cRecirculationPreValve_Event_Table



begin cPreValve_Event_Table
cPreValve=open; smSystem=sFastFill,cRecirculationPreValve=open; -; cPreValve=closed;
end cPreValve_Event_Table

begin cOutboardFillDrain_Event_Table
cOutboardFillDrain=closed; smSystem=sChilldown,mTimeState=60; -; cOutboardFillDrain=open;
end cOutboardFillDrain_Event_Table

begin cMainFillRedu_Event_Table
cMainFillRedu=closed; smSystem=sFastFill,cPreValve=closed; -; cMainFillRedu=open;
cMainFillRedu=open; smSystem=sTopping,mVolumeState=100; -; cMainFillRedu=closed;
end cMainFillRedu_Event_Table

begin cMainFill_Event_Table
cMainFill=closed; smSystem=sChilldown,mTimeState=165; -; cMainFill=open;
cMainFill=open; smSystem=sTopping,mVolumeState=100; -; cMainFill=closed;
end cMainFill_Event_Table

begin cInboardFillDrain_Event_Table
cInboardFillDrain=open; smSystem=sFastFill,cPreValve=closed; -; cInboardFillDrain=closed;
end cInboardFillDrain_Event_Table

begin cHighPointBleed_Event_Table
cHighPointBleed=closed; smSystem=sSlowFill,cChilldown=open,cTopping=open; -; cHighPointBleed=open;
end cHighPointBleed_Event_Table

begin cExternalTankVent_Event_Table
cExternalTankVent=closed; smSystem=sChilldown,mTimeState=60; -; cExternalTankVent=open;
cExternalTankVent=closed; smSystem=sFastFill,mVolumeState=98; -; cExternalTankVent=open;
cExternalTankVent=open; smSystem=sChilldown,mPressureState=43.7; -; cExternalTankVent=closed;
end cExternalTankVent_Event_Table

begin cChilldown_Event_Table
cChilldown=closed; smSystem=sChilldown,mTimeState=165; -; cChilldown=open;
cChilldown=closed; smSystem=sSlowFill; -; cChilldown=open;
cChilldown=open; smSystem=sChilldown,mPressureState=43.7; -; cChilldown=closed;
end cChilldown_Event_Table
```

B.2 Alternating Bit Protocol (transition function charts on page 13)

B.2.1 Input for Translation to NAYO Graph

```
begin sender
q0_s;
q1_s;
q2_s;
q3_s;
q4_s;
q5_s;
q0_s; -; mesg0; q1_s;
q1_s; ack1; -; q0_s;
q1_s; ack0; -; q2_s;
q2_s; -; mesg1; q3_s;
q3_s; ack0; -; q5_s;
q3_s; ack1; -; q4_s;
q4_s; -; mesg0; q1_s;
q5_s; -; mesg1; q3_s;
end sender

begin receiver
q0_r;
q1_r;
q2_r;
q3_r;
q4_r;
q5_r;
q0_r; mesg1; -; q1_r;
q0_r; mesg0; -; q2_r;
q1_r; -; ack1; q3_r;
q2_r; -; ack0; q0_r;
q3_r; mesg0; -; q4_r;
q3_r; mesg1; -; q5_r;
q4_r; -; ack0; q0_r;
q5_r; -; ack1; q3_r;
end receiver
```

B.3 TCP Protocol (original finite-state machine diagram shown in Figure B.1)

B.3.1 Input for Translation to NAYO Graph

```
begin tcp
closed;
syn_sent;
syn_rcvd;
listen;
established;
```

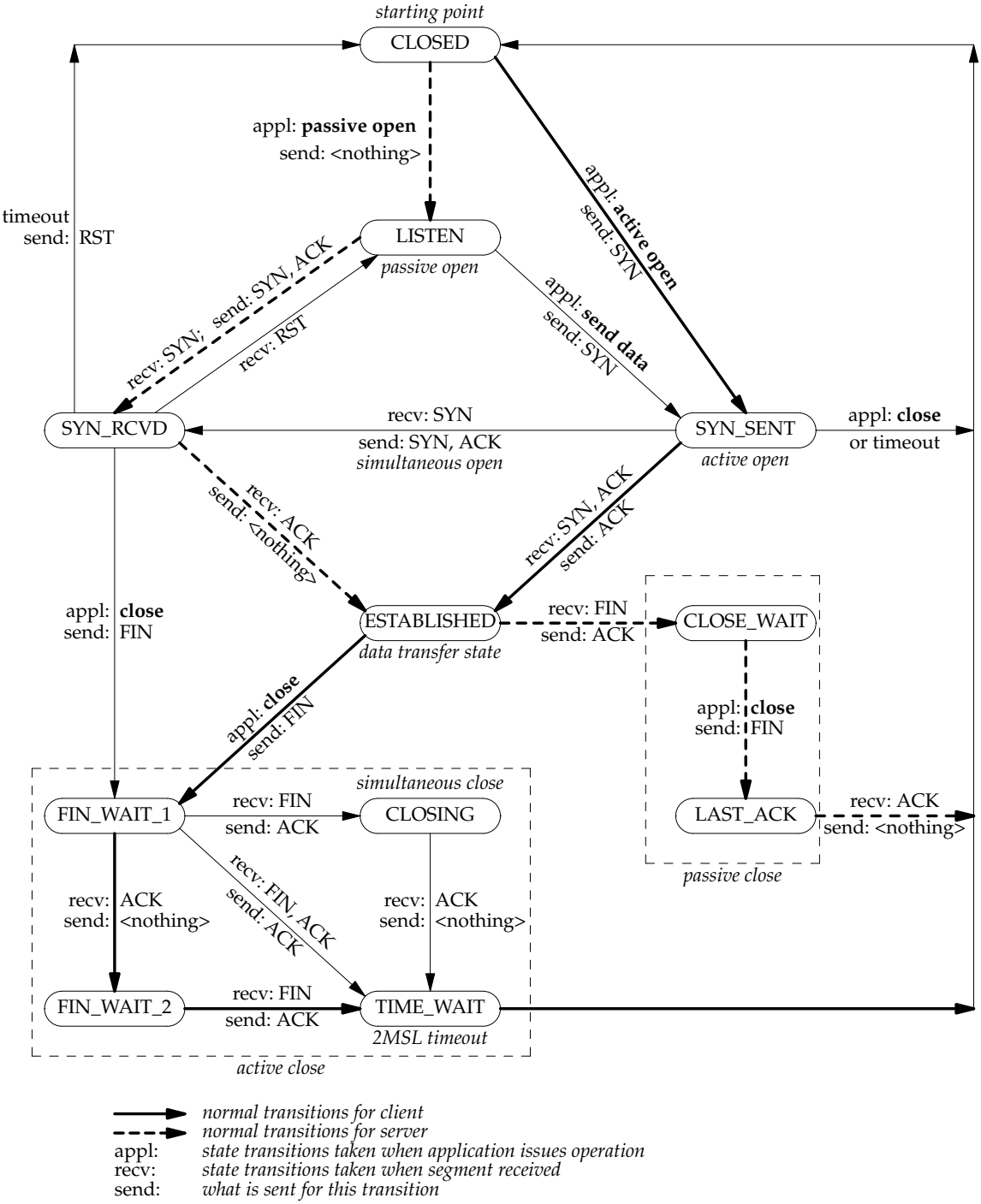


Figure B.1: TCP State Transition Diagram from [24].

```
fin_wait_1;
closing;
fin_wait_2;
time_wait;
close_wait;
last_ack;
closed; app_active_open; syn; syn_sent;
closed; app_passive_open; -; listen;
syn_sent; app_close; -; closed;
syn_sent; time_out; -; closed;
syn_sent; SYN; syn,ack; syn_rcvd;
syn_sent; SYN,ACK; ack; established;
syn_rcvd; rst; -; listen;
syn_rcvd; ACK; -; established;
syn_rcvd; app_close; fin; fin_wait_1;
listen; app_send_data; syn; syn_sent;
listen; SYN; syn,ack; syn_rcvd;
established; app_close; fin; fin_wait_1;
established; FIN; ack; close_wait;
fin_wait_1; FIN; ack; closing;
fin_wait_1; FIN,ACK; ack; time_wait;
fin_wait_1; ACK; -; fin_wait_2;
closing; ACK; -; time_wait;
fin_wait_2; FIN; ack; time_wait;
time_wait; time_out'; -; closed;
close_wait; app_close; fin; last_ack;
last_ack; ACK; -; closed;
end tcp

begin TCP
CLOSED;
SYN_SENT;
SYN_RCVD;
LISTEN;
ESTABLISHED;
FIN_WAIT_1;
CLOSING;
FIN_WAIT_2;
TIME_WAIT;
CLOSE_WAIT;
LAST_ACK;
CLOSED; APP_ACTIVE_OPEN; SYN; SYN_SENT;
CLOSED; APP_PASSIVE_OPEN; -; LISTEN;
SYN_SENT; APP_CLOSE; -; CLOSED;
SYN_SENT; TIME_OUT; -; CLOSED;
SYN_SENT; syn; SYN,ACK; SYN_RCVD;
SYN_SENT; syn,ack; ACK; ESTABLISHED;
SYN_RCVD; RST; -; LISTEN;
SYN_RCVD; ack; -; ESTABLISHED;
SYN_RCVD; APP_CLOSE; FIN; FIN_WAIT_1;
LISTEN; APP_SEND_DATA; SYN; SYN_SENT;
LISTEN; syn; SYN,ACK; SYN_RCVD;
ESTABLISHED; APP_CLOSE; FIN; FIN_WAIT_1;
ESTABLISHED; fin; ACK; CLOSE_WAIT;
```

```
FIN_WAIT_1; fin; ACK; CLOSING;
FIN_WAIT_1; fin,ack; ACK; TIME_WAIT;
FIN_WAIT_1; ack; -; FIN_WAIT_2;
CLOSING; ack; -; TIME_WAIT;
FIN_WAIT_2; fin; ACK; TIME_WAIT;
TIME_WAIT; TIME_OUT'; -; CLOSED;
CLOSE_WAIT; APP_CLOSE; FIN; LAST_ACK;
LAST_ACK; ack; -; CLOSED;
end TCP
```

Bibliography

[1] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[2] Bemina Atanacio. Modeling the Space Shuttle Liquid Hydrogen Subsystem. Technical report, The Software Engineering Institute, Carnegie Mellon University, 2000.

[3] A. Bertolino and L. Strigini. On the use of testability measures for dependability assessment. *IEEE Transactions on Software Engineering*, 20(12), 1994.

[4] Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency Checking of SCR-Style Requirements Specifications. In *International Symposium on Requirements Engineeing*, York, England, 1995.

[5] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models*. Benjamin / Cummings Publishing Company, Inc., 1986.

[6] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite-State Machines—A Survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996. Available at `http://citeseer.nj.nec.com/lee96principles.html`.

[7] Edmund A. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[8] Norman Fenton. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing, 1997.

[9] Jr. Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1995. This is the twentieth anniversary edition and includes chapters not published in the original.

[10] Gerard J. Holzmann. Basic SPIN Manual. Available at `http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.htm`.

[11] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[12] Henry Kautz and Bart Selman. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In *13th National Conference on Artificial Intelligence*, Portland, OR, 1996.

[13] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990. Available at `http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html`.

[14] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Lauback, Corina S. Păsăreanu, Robby, and HongJun Zheng. Bandera: Extracting Finite-State Models from JAVA Source Code. In *ICSE 2000*, pages 439–448, Limerick, Ireland, 2000. Available at `http://citeseer.nj.nec.com/corbett00bandera.html`.

[15] James M. Crawford and Andrew B. Baker. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI–94)*, volume 2, 1994.

[16] K. Karoui, A.Ghedamsi, and R. Dssouli. Study of Some Influencing Factors in Testability and Diagnostics Based on FSMs, 1996. Available at `http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html`.

[17] K. Karoui and R. Dssouli. Testability Analysis of the Communication Protocols Modeled by Relations, 1996. Avaliable at `http://citeseer.nj.nec.com/147902.html`.

[18] Michael A. Friedman and Jeffrey M. Voas. *Software Assessment: Reliability, Safety, Testability*. John Wiley & Sons, 1995.

[19] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2000.

[20] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[21] Patrice Godefroid. Model Checking for Programming Languages Using Verisoft. In *The Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, 1997. Available at `http://citeseer.nj.com/godefroid97model.html`.

[22] Peter Cheeseman, Bob Kanesfy, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence IJCAI–91, Sidney, Austrailia*, 1991.

[23] Richard Hamlet. Random Testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994. Available at `http://citeseer.nj.nec.com/hamlet94random.html`.

[24] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[25] Tim Menzies and Bojan Cukic. Intelligent Testing Can Be Very Lazy. In *Submitted to The First International Workshop on Intelligent Software Engineering*, Orlando, FL, 1999.

[26] Tim Menzies and Bojan Cukic. Adequacy of Limited Testing for Knowledge-Based Systems. *International Journal on Artificial Intelligence Tools*, 9(1), 2000.

[27] Tim Menzies and Bojan Cukic. Maintaining Maintainability = Recognizing Reachability. In *International Workshop on Empirical Studies of Software Maintenance (Wess 2000)*, San Jose, CA, 2000.

[28] Tim Menzies and Bojan Cukic. When to Test Less. *IEEE Software*, pages 107–112, 2000.

[29] Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing Nondeterminate Systems. In *ISSRE 2000*, San Jose, CA, 2000.

[30] Tim Menzies and Christoph C. Michael. Fewer Slices of PIE: Optimising Mutation Testing via Abduction. In *SEKE '99*, 1999. Available at `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-007.pdf`.

[31] Tim Menzies and Harhsinder Singh. Many Maybes Mean (Mostly) the Same Thing. In *2nd International Workshop on Soft Computing Applied to Software Engineering*, Netherlands, 2001.

[32] Tim Menzies and Paul Compton. The (Extensive) Implications of Evaluation on the Development of Knowledge-Based Systems. In *8th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1995.

[33] Tim Menzies and Paul Compton. Applications of Abduction: Hypothesis Testing of Neuroendocrinological Qualitative Compartmental Models. *Artificial Intelligence in Medicine*, pages 145–175, 1997.

[34] Tim Menzies and Ying Hu. Constraining Discussions in Requirements Engineering via Models. In *First International Workshop on Model-Based Requirements Engineering*, San Diego, CA, 2001.